BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

A CONVERTER BETWEEN THE LESS AND SASS STYLESHEET FORMATS
PŘEKLADAČ MEZI FORMÁTY LESS A SASS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR   ATTILA VEČEREK
AUTOR PRÁCE

SUPERVISOR   Prof. Ing. TOMÁŠ VOJNAR, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2016

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems   Academic year 2015/2016

Bachelor Project Specification

For: **Večerek Attila**
Branch of study: Information Technology
Title: **A Converter between the LESS and SASS Stylesheet Formats**
Category: Compiler Construction

Instructions for project work:
1. Get acquainted with the LESS and SASS dynamic stylesheet languages and study their differences.
2. Design and implement parsers from both languages into abstract syntax trees.
3. Design and implement code generators for both languages.
4. Verify correctness of the obtained converters through automated tests.
5. Evaluate results of your project and discuss possible extensions of the implemented converters.

Basic references:
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, Addison Wesley, 2006.
- Hixon, J.: An Introduction To LESS, And Comparison To Sass, In: Smashing Magazine, 2011. Available online: http://www.smashingmagazine.com/2011/09/an-introduction-to-less-and-comparison-to-sass/.
- Getting started: An overview of Less, how to download and use, examples and more. Available online: http://lesscss.org/.

Requirements for the first semester:
   No requirements.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

   The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

   Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 18, 2016

Petr Hanáček
Associate Professor and Head of Department

Abstract

The aim of this thesis is to research the differences between the CSS preprocessor languages, namely Less and Sass, and find applicable transformation methods to implement a converter between their dynamic stylesheet formats. A general introduction to the concept of CSS preprocessors is provided first, which is followed by a thorough description of the Less and Sass language features. In addition to this, all the discovered differences are stated and illustrative examples of the invented conversion methods are provided in this work. This is followed by the description of the design and implementation of the proposed converter. As a part of the contribution of this thesis, a CSS comparison tool based on abstract syntax tree transformation has also been developed. Its design is described along the testing procedure used to verify the invented conversion methods. The last part of the work summerizes the achieved results and the future directions of the converter.

Abstrakt

Cílem této bakalářské práce je výzkum rozdílů mezi CSS preprocesorovými jazyky, jmenovitě Less a Sass, a nalezení použitelných transormačních metod k implementaci překladače mezi jejich formáty. Nejprve je předložen koncept CSS preprocesorů a následuje detailní popis vlastností jazyků Less a Sass. V této práci jsou uvedené všechny zjištěné rozdíly, a pak jsou představeny nové konverzní metody s demonstrativními příklady. Následuje popis návrhu a implementace překladače. Součástí této práce je tvorba nástroje pro porovnávání CSS, který je postaven na základě transformace abstraktního syntaktického stromu. Návrh komparátoru je popsán spolu s procesem testování, jenž byl použitý pro verifikaci zavedených konverzních metod. V poslední části práce jsou shrnuty dosažené výsledky a je navržen budoucí vývoj překladače.

Keywords

CSS preprocessor, Less, Sass, dynamic stylesheets, abstract syntax tree transformation, CSS comparison

Klíčová slova

CSS preprocesor, Less, Sass, dynamické styly, transformace abstraktních syntaktických stromů, porovnávání kaskádových stylů

Reference

VEČEREK, Attila. *A Converter between the LESS and SASS Stylesheet Formats*. Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

A Converter between the LESS and SASS Stylesheet Formats

Declaration

I hereby declare that the present bachelor's thesis was composed by myself under the leadership of Bc. Dávid Halász and that the work contained herein is my own. On behalf of the Faculty of Information Technology, the thesis has been formally led by Prof. Ing. Tomáš Vojnar, Ph.D. I have acknowledged all the sources of information which have been used in the thesis.

<div align="right">

. . . .
Attila Večerek
May 17, 2016

</div>

Acknowledgements

I would like to thank Bc. Dávid Halász for being a supportive supervisor and Prof. Ing. Tomáš Vojnar, Ph. D. for his expert advice on content editing and typographical arrangements of this work.

Contents

Chapter 1

Introduction

Cascading Style Sheet (CSS in short) is a declarative programming language that describes the way HTML elements are to be displayed on screen, paper, or in other media. The larger a CSS stylesheet becomes, the more difficult it is to maintain. To overcome those maintenance issues, CSS preprocessors have been developed. Many of the popular CSS frameworks, like Bootstrap and Foundation, have been built using these tools.

Preprocessors like Less, Sass and Stylus have become very popular among web developers. Currently, many of the developers who formerly preferred Less are moving to Sass. The same applies to some of the CSS frameworks that were developed using Less, i.e. Bootstrap. To facilitate and automate the transition from Less to Sass, a converter between these stylesheet formats has to be created.

One of the companies that also deal with the challanges of the transition is Red Hat, Inc. — the world's leading provider of open source solutions — which also proposed and expertly led this work. In view of the foregoing, the aim of this thesis is to create a converter between the dynamic stylesheet formats Less and Sass, which could potentially replace the existing solutions in use. While working on the objectives, it was necessary to get fully acquainted with the CSS preprocessors and their programming languages as well as study the differences between them. Conversion methods were found and documented, which can also be used as a guide for manual transition of Less projects to Sass. The existing converters work on a string replacement basis using regular expressions. Experience has shown that these converters do not return valid results. In practice, this means that developers still have to rewrite the output stylesheet after the conversion. Our goal is to eliminate or at least minimize the necessary work after the conversion. In order to achieve this goal, our converter was designed to use abstract syntax tree transformation methods. In the converter, we use the engines of the respective CSS preprocessors for parsing the input and generating the output stylesheets. Finally, as part of this work a CSS comparison tool was also created to verify the implemented converter. The rest of the thesis is structured as follows:

In *Chapter 2*, we will have a look at the CSS preprocessors in general and the reasons of their existence and need. Secondly, we will briefly characterize Less and Sass. Their main features will be introduced along with some real-world examples of their usage. Furthermore, we will deeply dive into their differences by enlisting all those features of Less that are problematic from the perspective of conversion. Finally, we will introduce the existing conversion possibilities that are currently being used in practice. *Chapter 3* describes the implementation, highlights the key steps of the conversion process and names the language

features, that are supported by the implemented converter. In *Chapter 4*, we will thoroughly describe the testing procedures that need to be undertaken in order to verify the accomplished results.

Chapter 2

CSS Preprocessors

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 [6]. CSS is a declarative language and lacks most of the traditional programming constructs, like variables and functions, which enable code reuse and structured programming. Having said that, CSS projects may turn difficult to scale and maintain over time, since web developers have to perform many of the same activities over and over again.

CSS preprocessors, also referred to as precompilers, have been introduced as superset languages to extend CSS by supporting those missing constructs [9]. Being an extension of CSS means that valid CSS code is valid preprocessor code, as well. A preprocessor, in the context of CSS, is a simple transformation tool that takes preprocessor-formatted file and outputs CSS code that browser understands. There are several existing CSS preprocessors, each introducing its own language. According to SitePoint — a popular online publisher of books, courses and articles for web developers — in 2014, the two most popular CSS preprocessors were Syntactically Awesome Style Sheets (henceforth, Sass) and Less.js (henceforth, Less) [8]. In the following two sections, we will have a brief look at their main characteristics.

2.1   Less

Less is an open source CSS preprocessor that was created in 2009 by Alexis Sellier. Originally, it was written in Ruby and later it was ported to JavaScript. Today, Less is maintained by a group of core contributors, with the massive support and involvement of the community. It runs inside the browser and the well-known cross-platform environments: Node.js and Rhino [1].

There are two variants of Less. The client-side variant compiles Less code in real-time in the browser. It only supports the recent versions of modern browsers (e.g., Chrome, Firefox, Safari and IE). However, its use in production is not recommended due to performance implications. For the best performance, it is recommended to compile the Less code on the server side. On the other hand, this feature comes handy if a developer wants to allow users to make modifications to a theme and show the result in real-time.

Less is also a declarative language backwards compatible with CSS and its extra features use existing CSS syntax. Those features turn the old way of developing stylesheets into a more scalable, extensible and maintainable process by applying the "Don't Repeat Yourself" (DRY) principle. Some of them will be introduced in the following parts of this

chapter. The definitions and examples used in this section were taken from the official documentation of the language features of Less [2].

2.1.1 Variables

The variables make it possible to control commonly used values in a single location. Their most significant use case is when defining color palettes for themes. It is a good practice to create a *colors.less* file containing all the color variables used in a project.

Scope

CSS is a declarative language and so is Less. Note that variables in Less have a behaviour much closer to constants since they can only be defined once per execution context. Variables are lazy loaded and do not have to be declared before being used. When defining a variable twice, the last definition of the variable is used, searching from the current scope upwards. This is similar to CSS itself where the last property inside a definition is used to determine the value. This behaviour can be best described through the following code example.

```
.lazy-eval-scope {
  width: @var;  // @var: 9%
  @a: 9%;
}

@var: @a;  // @var: 100%
@a: 100%;
```

Example 2.1: Less: Lazy loading

As we can see in Example 2.1, the variable `@var` in the global execution context holds the value of 100%, whereas the one that is defined in the scope of the *.lazy-eval-scope* selector holds the value of 9%. This is caused by the redefinition of `@a` in the local scope.

2.1.2 Extend

Extend is a Less pseudo-class which merges the selector it is put on with ones that match what it references. It can be either attached to a selector or placed into a ruleset. It accepts a selector as a parameter optionally followed by the keyword *all*. When you specify the *all* keyword last in an extend argument it tells Less to match that selector as part of another selector. Simply said, it will apply the extend on all selectors that contain the selector sequence specified as the parameter.

In Example 2.2, extend creates new selectors by replacing the element `.test` with `.replacement`. Those newly created selectors are attached to the original list of selectors containing the specified selector sequence. The output of the example Less code is shown in Example 2.3.

The classic use case of extend is to avoid using a base class in the HTML template and thus making selector inheritance possible just like the class inheritance we know from

object-oriented programming. Assume the following example. There is a class *bear* and its superclass *animal*. In HTML, the use of class bear would be subject to use of the animal class, as well: **<a class**="animal bear">Bear</**a**>. However, using the extend feature, it is possible to simplify the html and use the class *bear* without its superclass. It can be achieved with the Less code found in Example 2.4.

```less
// extend_all.less
.a.b.test,
.test.c {
  color: orange;
}

.replacement:extend(.test all) {}
```

Example 2.2: Less: Extend 'all'

```css
// extend_all.css
.a.b.test,
.test.c,
.a.b.replacement,
.replacement.c {
  color: orange;
}
```

Example 2.3: Output: Extend 'all'

```less
// animals.less
.animal {
  background-color: black;
  color: white;
}
.bear {
  &:extend(.animal);
  background-color: brown;
}
```

Example 2.4: Less: Classic use case of extend

2.1.3 Mixins

Mixins are one of those language features provided by Less that keep the code *DRY*, thus producing CSS in a more efficient way. Less provides different types of mixins for slightly

7

different use cases but all share the same purpose. Basically, there is no syntactical difference between a mixin definition and a selector declaration. Mixins are called by specifying the name of the selector followed by optional parentheses and a mandatory semicolon as shown in Example 2.5.

```less
// mixins.less
.a {
  color: red;
}
#b() {
  color: black;
}
.mixin-class {
  .a;
}
#mixin-id {
  #b();
}
```

Example 2.5: Less: Mixins

```css
// mixins.css
.a {
  color: red;
}
.mixin-class {
  color: red;
}
#mixin-id {
  color: black;
}
```

Example 2.6: Output: Mixins

Parentheses are optional to use in mixin definitions and they serve as a flag meaning that the mixin should not be output in the final CSS code as a selector rule as shown in Example 2.6.

Mixin scope

It is important to understand the way mixins are placed in the scope and which variables they have access to. If a variable is not available in the mixin, Less will look for its value in its parent scopes until it reaches the global execution context as seen in Example 2.7. If the value cannot be found even there, Less will start searching from the caller scope until it hits the global execution context again. The latter case can be seen in Example 2.8.

```less
@x: 1;
.mixin() {
  z-index: @x;
}
#namespace {
  @x: 2;
  .mixin();
  // outputs: z-index: 1;
}
```

Example 2.7: Variable in parent scope

```less
.mixin() {
  z-index: @x;
}
#namespace {
  @x: 2;
  .mixin();
  // outputs: z-index: 2;
}
```

Example 2.8: Variable in caller scope

8

Parametric Mixins

Mixins can also take arguments, which are variables passed to the block of selectors when they are called. Less also provides so called named parameters. It means that any parameter can be referenced by its name without following any special order as shown in Example 2.9.

```less
.mixin(@color: black; @margin: 10px; @padding: 20px) {
  color: @color;
  margin: @margin;
  padding: @padding;
}
.class1 {
  .mixin(@margin: 20px; @color: #33acfe);
}
.class2 {
  .mixin(#efca44; @padding: 40px);
}
```

Example 2.9: Less: Parametric mixins using named parameters

Pattern-matching Sometimes developers want to change the behaviour of a mixin, based on the parameters passed to it. This feature is known as method overloading in object-oriented programming. In Less, it can be achieved by its pattern-matching feature. An example of pattern-matching and its CSS output can be seen in Example 2.10 and 2.11 respectively.

```less
.mixin(dark; @color) {
  color: darken(@color, 10%); // color: #6f6f6f
}
.mixin(light; @color) {
  color: lighten(@color, 10%); // color: #a2a2a2
}
.mixin(@_; @color) {
  display: block;
}
@switch: light;

.class {
  .mixin(@switch; #888);
}
```

Example 2.10: Less: Pattern-matching

As seen in the output, the color passed to *.mixin* was lightened and the class also received the `display: block;` declaration. The method overloading in Less behaves slightly

9

different that we are used to. Instead of calling one specific mixin, all the mixins are called that match the pattern. The underscore variable (@_) plays the role of a wildcard, accepting any parameter.

```
.class {
  color: #a2a2a2;
  display: block;
}
```

Example 2.11: Output: Pattern-matching

Mixins as functions

Variables and mixins defined inside a mixin are accessible from the caller scope. However, they will be overridden by the locally defined or already "mixed in" variables and mixins with the same name. Only the variables present in caller scopes are protected. The variable inherited from parent scopes are overridden as shown in Example 2.12 and 2.13.

```
.mixin() {
  @size: 20px;
  @definedOnlyInMixin: 20px;
}

.class { //caller scope
  margin: @size @definedOnlyInMixin;
  .mixin();
}

@size: 15px; // callers parent scope - no protection
```

Example 2.12: Less: Mixins as funcions

```
.class {
  margin: 20px 20px;
}
```

Example 2.13: Output: Mixins as funcions

Passing rulesets to mixins

A detached ruleset is a group of CSS properties, nested rulesets, media declarations or anything else stored in a variable. It can be included into a ruleset or other structures with its properties being copied into the callers scope. It can be also used as a mixin argument and pass it around as any other variable. As opposed to mixins, the parentheses after a detached ruleset call are mandatory. It is useful when defining a mixin that abstracts out a piece of code in a media query or a non-supported browser class name.

Example 2.14 shows the definition and use of a detached ruleset. Its output CSS can be seen in Example 2.15.

```
@my-ruleset: {
  .my-selector {
    @media tv {
      background-color: black;
    }
  }
};
@media (orientation:portrait) {
  @my-ruleset();
}
```

Example 2.14: Less: Detached ruleset

```
@media (orientation: portrait) and tv {
  .my-selector {
    background-color: black;
  }
}
```

Example 2.15: Output: Detached ruleset

Scope   The scoping is also interesting in case of detached rulesets. Hence, it can also access the variables and mixins defined in its caller scope. If both scopes contain the same variable or mixin, declaration scope value takes precedence as seen in Example 2.16. *Declaration scope* is the one where the body of detached rulesets is defined.

```
@detached-ruleset: {
  caller-variable: @caller-variable; // variable is undefined
      here
  .caller-mixin(); // mixin is undefined here
};

selector {
  // use detached ruleset
  @detached-ruleset();

  // define variable and mixin needed inside the detached ruleset
  @caller-variable: value;
  .caller-mixin() {
    variable: declaration;
  }
}
```

Example 2.16: Less: Definition and caller scope visibility

2.1.4 Import Directive

Less can import styles from other style sheets. `@import` statements may be treated differently by Less depending on the file extension and speciefied import options. Only files having a *.css* extension will be treated as CSS imports. In all other cases, the file will be treated as a Less file.

2.1.5 Mixin Guards

Mixin guards are used as conditional mixins. Less has opted to implement conditional execution of mixins via **guards** instead of *if/else* statements, in the vein of `@media` query feature specifications. Example 2.17 shows three mixin definitions of which two are guarded.

```
.mixin (@a) when (lightness(@a) >= 50%) {
  background-color: black;
}
.mixin (@a) when (lightness(@a) < 50%) {
  background-color: white;
}
.mixin (@a) {
  color: @a;
}
.class1 { .mixin(#ddd) }
.class2 { .mixin(#555) }
```

Example 2.17: Less: Mixin guards

Additionally, the `default` function may be used as an alternative to *else* as shown in Example 2.18.

```
.mixin (@a) when (@a > 0) { ...  }
.mixin (@a) when (default()) { ... } // matches only if first
    mixin does not, i.e. when @a <= 0
```

Example 2.18: Less: Default guard

2.1.6   CSS Guards

CSS guards play the same role as mixin guards, the only difference is that they are applied to CSS selectors instead of mixins.

2.1.7   Loops

In Less a mixin can call itself. Such recursive mixins can be achieved with loops combined with guard expressions and pattern matching (see Section 2.1.3). A generic example of using recursive loops is the generation of CSS grid classes as shown in Example 2.19 and 2.20.

```
.generate-columns(4);

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

Example 2.19: Less: Loops

```
.column-1 { width: 25%; }
.column-2 { width: 50%; }
.column-3 { width: 75%; }
.column-4 { width: 100%; }
```

Example 2.20: Output: Loops

2.1.8 Merge

The `merge` feature allows for aggregating values[1] from multiple properties into a comma or space separated list under a single property. It is useful for properties such as background and transform.

2.1.9 Parent Selectors

The `&` operator represents the parent selector of a nested rule and is most commonly used when applying a modifying class[2] or pseudo-class to an existing selector.

2.1.10 JavaScript Evaluation

Less is capable of evaluating JavaScript code used in its dynamic stylesheets. However, this feature is not longer documented on the official page of the CSS preprocessor. In order to use this feature, the JavaScript code has to be enclosed in backticks as shown in Example 2.21

```
// @var = "HELLO WORLD!"
@var: `"hello world".toUpperCase() + '!'`;
```

Example 2.21: Less: JavaScript Evaluation

2.2 Sass

Sass is an open source project proposed by Hampton Catlin. However, the primary developer and architect of Sass is Natalie Weizenbaum, a software developer at Google. The second lead developer worth to mention is Chris Eppstein, creator of Compass, the first Sass-based framework.

The project is implemented in Ruby, although there is a C/C++ port of the Sass engine created by the original author for efficiency and higher portability [4]. As opposed to Less, Sass can be compiled only on the server-side. It can be used in three ways: as a command-line tool, as a standalone Ruby module, and as a plugin for any Rack-enabled framework, including *Ruby on Rails* and *Merb*.

Sass also offers two syntaxes to choose from. The new main syntax is known as SCSS (for "Sassy CSS"), and it is a superset of the CSS syntax, meaning that every valid CSS stylesheet is valid SCSS, as well. SCSS files use the extension `.scss`.

The second, older syntax is known as Sass — the indented syntax. It is inspired by Haml — the markup language used to cleanly and simply describe HTML templates [5] — and it is intended for people who prefer conciseness over similiarity to CSS. Files in the indented syntax use the extension `.sass`.

Sass is able to update the generated CSS every time the original Sass source code changes and has a strong *caching* feature implemented. By default, Sass caches compiled

---

[1] http://lesscss.org/features/#merge
[2] http://lesscss.org/features/#parent-selectors-feature

templates and partials, meaning faster re-compilation of large collections of Sass files. The CSS style being output can be also chosen. Sass offers four different output styles - *nested*, *expanded*, *compact* and *compressed*.

In spite of Sass being an imperative language, it shares a lot of features with Less that work in a very similar manner. They both support the following features:

- nested rules,

- parent selectors,

- variables,

- variable interpolations,

- import directives,

- extend directives and

- mixins.

The following sections will describe those features of Sass that are unique in terms of comparison to Less. The definitions and examples were taken from the official documentation of Sass [3].

2.2.1 Placeholder Selectors

Sass supports a special type of selector called a "placeholder selector". They are meant to be used with the `@extend` directive. Their purpose is to generate leaner CSS than the one that would be output using classic selectors instead, since rulesets that use placeholder selectors will not be rendered to CSS. The behaviour of this feature is similar to the mixins used with parentheses in Less.

2.2.2 Nested Properties

CSS possesses a few properties that closely relate to each other for instance, `font-family`, `font-size` and `font-weight` are all in the `font` "namespace". Sass provides a shortcut for not having to type out all the closely related properties during their declaration. This feature is shown in Example 2.22.

```
.funky {
  font: {
    family: fantasy;
    size: 30em;
    weight: bold;
  }
}
```

Example 2.22: Sass: Nested properties

2.2.3 @at-root Directive

The `@at-root` directive causes one or more rules to be emitted at the root of the document, rather than being nested beneath their parent selectors. By default, `@at-root` just excludes selectors. However, it is also possible to use it to move outside of nested directives such as `@media` or `@supports`. According to the CSS output shown in Example 2.24, the declaration `color: red;` has been moved to the root of the document encased in its original selector and `@supports` directive without the `@media` directive as specified in Example 2.23.

```
@media print {
  @supports(transform-style: preserve-3d) {
    #id, .page {
      width: 8in;
      @at-root(without: media) {
        color: red;
      }
    }
  }
}
```

Example 2.23: Sass: @at-root directive

```
@media print {
  @supports (transform-style: preserve-3d) {
    #id, .page {
      width: 8in;
    }
  }
}
@supports (transform-style: preserve-3d) {
  #id, .page {
    color: red;
  }
}
```

Example 2.24: Output: @at-root directive

2.2.4 @debug, @warn and @error

Sass also offers a possibility to print values, expressions and messages to the standard error output stream using three different directives.

The `@debug` directive is useful for debugging Sass files that use complicated language constructs. The `@warn` directive prints the value of a Sass expression to the output stream. It is useful for warning users of deprecations or recovering from minor mixin usage mistakes. The main distinction between `@warn` and `@debug` is that a stylesheet trace will be printed

out along with the message showing the origin of the warning. The `@error` directive throws a fatal error, including the stack trace. Its usefullness lies in validating arguments to mixins and functions. However, there is currently no possibility to catch errors.

2.2.5 Control Directives

Sass supports basic control directives and expressions for including styles only under some conditions or including the same style several times with variations. Opposed to Less guards (see Section 2.1.4), Sass implemented control directives to use standard keywords and constructs.

if()

The built-in `if()` function returns only one of two specified values based on the given condition as can be seen in Example 2.25. It can be used as an alternative to ternary operators.

```
if(true , 1px , 2px) => 1px
if(false , 1px , 2px) => 2px
```

Example 2.25: Sass: if() function

@if

The `@if` directive takes an expression and uses the rules nested beneath it if the expression evaluates to `true`. It can be followed by several `@else if` statements and one final `@else` statement. This works just like the standard *if-else-if* constructs we are all familiar with.

@for, @each and @while

While Less implements only one form of loop, Sass offers the flexibility to choose from three posible ways of implementing iterations.

The `@for` directive repeatedly ouputs a set of styles. For each repetition, a counter variable is used to adjust the output[3].

The `@each` directive loops through each item of an expression returning a list or a map and outputs [4] the rules it contains. The data types as list and map will be introduced later, in the section discussing the differences between Less and Sass 2.3.

The `@while` directive takes an expression and repeatedly outputs [5] the nested rules until the statement evaluates to `false`.

2.2.6 Function Directives

Sass implements function directives to return values instead of delegating this behaviour to mixins as opposed to Less — see Section 2.1.3. The functions can access any globally

---

[3]http://sass-lang.com/documentation/file.SASS_REFERENCE.html#_10
[4]http://sass-lang.com/documentation/file.SASS_REFERENCE.html#each-directive
[5]http://sass-lang.com/documentation/file.SASS_REFERENCE.html#_12

defined variable as well as accept arguments like mixins do. The `@return` statement must be called to set the return value of the function. It is recommended to prefix the functions to avoid naming conflicts. For historical reasons, function names as well as all other Sass identifiers can use hyphens and underscores interchangeably, thus referring to the same object.

Example 2.26 shows a function that uses globally defined variables to return the width of a grid. The output CSS is shown in Example 2.27.

```
$grid-width: 40px;
$gutter-width: 10px;

@function grid-width($n) {
  @return $n * $grid-width + ($n - 1) * $gutter-width;
}

#sidebar { width: grid-width(5); }
```

Example 2.26: Sass: Function directive

```
#sidebar {
  width: 240px; }
```

Example 2.27: Output: Function directive

2.3 Difference between Less and Sass

In this section, we will explore all the syntactic and semantic differences between Less and Sass that we were able to uncover during our studies of these languages. It will also describe the solutions we proposed to overcome those differences and thus create equivalent Sass representation of Less-specific features.

First, in order to have a solid ground for sound transformations between Less and Sass files, we need to define the equivalence of dynamic stylesheets written in those preprocessor languages.

Definition 2.1. A Sass file is an equivalent representation of a Less file if and only if the CSS output produced by the two files is equivalent.

In the following sections, we explain the syntactic and semantic differences between Less and Sass. Furthermore, we describe how to convert each of the introduced Less language features into its equivalent Sass representation. Examples of the conversion methods will also be provided.

2.3.1 Variables

The syntactic difference in declaring variables is the leading symbol used for indicating a variable. In Less, it is the `@` symbol, whereas in Sass it is the `$` sign. Another difference can be found in the naming rules of variables. Sass treats the dash (`-`) and underscore (`_`) in the names of variables and mixins as interchangeable.

The only semantic difference we could find is the ability to store different data types. Less is more dynamic from this perspective, since it allows rulesets to be stored in variables that can be passed to mixins as arguments. However, Sass implements its own way of passing rulesets to mixins.

The variables in both languages are capable of storing selectors. However, Sass throws a syntax error upon trying to store a selector starting with a dot (`.`) representing a class identifier. The solution to this problem is to use the built-in function of Sass called `unquote` that returns the unquoted form of any string value. It is shown in Example 2.28 and 2.29.

```
@variable: .bucket;
@{variable} {
  color: blue;
}
```

Example 2.28: Less: Variable storing class identifier

```
$variable: unquote(".bucket");
#{$variable} {
  color: blue;
}
```

Example 2.29: Sass: Variable storing class identifier

Variable defaults

Sass offers its users a way to assign values to variables in case they had not been set by applying the `!default` flag just after the assignment.

Variable interpolation

The syntactic difference of this feature can be also seen in Example 2.28 and 2.29. Regarding the semantic difference, Sass does not support dynamic imports and variable names. Hence, the variable interpolation cannot be used in *@import statements* and *variable names*. However, there is a solution for the latter incompatibility. The global scope should create a *map of variables* consisting of the variables defined inside of it using their *names* as keys and their *references* as values. All other scopes should create their own *local copy* of that global variable rewritten by the *merged map* of the global and local variable maps as shown in Example 2.31, which is the equal representation of the Less code shown in Example 2.30.

```
#id {
  @another_var: "This is the content.";
  @var: "another_var";
  content: @@var;
}
```

Example 2.30: Less: Variable variables

```
$__l2s__vv: ();

#id {
  $var: "another_var";
  $another_var: "This is the content.";
  $__l2s__vv: map-merge($__l2s__vv, (
    "var": $var,
    "another_var": $another_var
  ));

  content: map-get($__l2s__vv, $var);
}
```

Example 2.31: Sass: Variable variables

Variable naming

As earlier mentioned, the *dash* and *underscore* characters are treated the same way in variable, mixin and function identifiers by Sass. To avoid possible naming collision, the identifiers containing at least one of those characters need to be properly distinguished. In our opinion, the most straigth-forward solution is to double the first occurence of either the dash or the underscore character. It is important to choose only one of the interchangable characters, otherwise the possibility of naming collision would remain as shown in Example 2.32.

```
// collision possibility eliminated
@rh-light_green;    => $rh--light_green;
@rh_light_green;    => $rh_light_green;

//collision possibility remains
@rh-light_green;    => $rh--light__green;
@rh_light_green;    => $rh__light__green;
```

Example 2.32: Naming collision avoidance

2.3.2   Data Types

Both languages support the following main data types:

- numbers (e.g. 1.2, 13, 10px),

- strings of text, with and without quotes (e.g. ''foo", 'bar', baz),

- colors (e.g. blue, #04a3f9, rgba(255, 0, 0, 0.5)),

- list of values, separated by spaces or commas (e.g. 1.5em 1em 0 2em; Helvetica, Arial, sans-serif).

However, on both sides there are either unsupported or nondifferentiated data types present. For instance, Less does not differentiate `boolean` and `null` types from unquoted strings and it also does not support `map` values. On the other hand, the ability to store rulesets in variables is missing from Sass, where the CSS property values, such as *Unicode ranges* and *!important* declarations are not differentiated from unquoted strings. Less implements an individual prototype to distinguish the Unicode range from other values.

Numbers

There is a significant difference with respect to how Less and Sass handle units and number operations.

Sass supports unit-based arithmetic. Additionally, Sass implements conversion tables so that any comparable units can be combined. This means, that Sass doeas not operate on numbers with incompatible units and two numbers with the same unit that are multiplied together produce square units resulting in invalid CSS value as shown in Example 2.33. Sass throws an error on attempt to use such a value.

On the other hand, Less is more lenient regarding unit handling during number operations. It takes the first unit found in the expression and executes the expression as if the remaining numbers were unitless as shown in Example 2.34.

Both Less and Sass implement support for user defined units as a form of future proofing against changes in the W3C specification.

```
1cm * 1em => 1cm * em        // Error: invalid CSS value
2in * 3in => 6in*in          // Error: invalid CSS value
2in + 3cm + 2pc => 3.5144357in
3in / 2in => 1.5
```

Example 2.33: Sass: Unit handling

```
1cm * 1em => 1cm
2in * 3in => 6in
2in + 3cm + 2pc => 3.5144357in
3in / 2in => 1.5
```

Example 2.34: Less: Unit handling

Precision   Another difference in Less and Sass is the number precision they use to calculate with. By default, the number precision used by Less is 8 digits after the decimal point, whereas in Sass it is only 5. However, Sass lets the user define the precision used either programatically or through a command-line argument.

2.3.3   Scoping

The most significant difference between Less and Sass is the way that each of the compared languages handle scoping. According to Matthew Dean, a core member of the team behind developing Less, Sass is an **imperative** language, whilst Less is **declarative** [7]. Both languages are extensions to the CSS in terms of syntax. However, the programming paradigm used in these languages differs. While in a declarative language the programmer only declares the *actions* to perform, in an imperative language the *modus* and the *order* of those actions are also defined. There are two challenges to tackle in terms of finding a conversion method for this difference both originating from the lazy loading feature of Less.

1. Variables can be used *before their declaration.* If the given variable or mixin cannot be found in the caller scope, it will look for its value in the parent scopes until it is found or the lookup fails after reaching the end of the global scope. If a variable is defined multiple times in a given scope, the last definition will be used.

2. A variable is evaluated in place of its use according to its definition using the values of variables reachable from the caller scope. See Example 2.1, where the variable `@a` is redefined in the caller scope and thus its new value is used during the evaluation.

An equivalent representation in Sass can be achieved by performing the following two actions:

1. Change the order of variable declarations in each scope, so the declaration always precedes its use.

2. A variable can reference other variables in its declaration, whose value can be overridden in local scopes. In such cases, the variable needs to be redeclared in those local scopes with its original definition as shown in Example 2.35.

```
$a: 100%;
$var: $a;

.lazy-eval-scope {
  $a: 9%;
  $var: $a;

  width: $var;
}
```

Example 2.35: Sass: Lazy loading solution

22

2.3.4   Extend

There is a syntax-related difference in the way the `extend` feature is used in the languages being compared. While in Less, it is used like a CSS placeholder, in Sass this feature is represented by a standalone directive. Less presents two ways to use the extend:

- attached to a selector - it is possible to use multiple extends following each other,

- inside a ruleset - however, still being attached to the parent selector (`&` symbol) as seen in Example 2.4.

On the other hand, Sass uses the directive always inside of a ruleset. If there are multiple attached extends to a selector, Sass would place all of them inside of the ruleset.

Exact matching

By default, Less looks for exact match between selectors when applying its extend feature. However, it will match selectors regardless of containing single quotes, double quotes, or no quotes as shown in Example 2.36.

```
// the following extends will not match the specified selectors
:extend(a:hover:visited)        => a:visited:hover
:extend(.class)                 => *.class
:extend(:nth-child(n+3))        => :nth-child(1n+3)

// however, the following extend will match all the specified
    selectors
:extend([title='identifier'])   => [title='identifier'],
                                   [title="identifier"],
                                   [title=identifier]
```

Example 2.36: Less: Extend only exact matching selectors

Sass, on the other hand, understands the logic behind CSS selectors and is able to tell, whether two selectors are equal, even if the match is not exact. To create an equivalent Sass representation for such situations — also seen in Example 2.37 — it is necessary to remove the affected extends from the Sass code as shown in Example 2.38.

```
.a.class {
  color: blue;
}
.test:extend(.class) {} // this will NOT match the selector above
```

Example 2.37: Less: Exact match not found

```
.a.class {
  color: blue;
}
.test {} // extend should be removed, not an exact match
```

Example 2.38: Sass: Exact match not found

Multiple extended selectors

In Less, each member of a selector list can have one or multiple *extend* pseudo-classes
attached to it. Since Sass does not support such a language construct, it is needed to refactor
such a list of selectors to achieve equal CSS output. It can be achieved by segragating the
list of selectors into separate declaration rules of those selectors that have at least one
extend pseudo-class attached to it. All extends should be put inside the ruleset. To the
remaining list of selectors — even if it is empty — a placeholder selector (see Section 2.2.1)
should be appended. Afterwards, the created placeholder selector should extend all the
segregated selectors, as demonstrated in Example 2.39 and 2.40.

```
.big-division:extend(.division),
.big-bag,
.big-bucket:extend(.bag):extend(.bucket) {
  // body
}
```

Example 2.39: Less: Multiple extended selectors

```
.big-division {
    @extend .division;
    @extend %_.big-bag;
}
.big-bucket {
    @extend .bag;
    @extend .bucket;
    @extend %_.big-bag;
}
.big-bag, %_.big-bag {
  // body
}
```

Example 2.40: Sass: Multiple extended selectors

Placeholder selector   Note that placeholder selector names must start with the **%** char-
acter which cannot be followed by the **#** nor **.** characters. In addition to these, it cannot

24

contain a * nor whitespace characters. All other characters also accepted by selectors can be used when generating placeholder names.

Extending nested selectors

Less is able to match and extend nested selectors, as opposed to Sass. In order to create the equal representation of this feature, a new placeholder selector should be created and used to extend the nested selector. The ruleset of the nested selector should be copied into the placeholder selector as seen in Example 2.41.

The placeholder name should be generated out of the selector list having the aforementioned rules in mind. The prohibited characters should be escaped by replacing them with a chosen sequence of characters. This cannot lead to naming collisions, since placeholders are not specified by Less.

```
* #a .class {
  color: blue;
}
%__uni__s#a__s.class {
  // placeholder name generated from the selector list
  color: blue;
}
a {
  @extend %__uni__s__#a__s.class;
}
```

Example 2.41: Sass: Extending nested selectors

Extend inside @media rulesets

Based on the language feature specification of both languages, this feature should work the same way. However, we found a case, where Sass fails its defined behaviour. According to the Sass language specification, the use of *@extend* within *@media* (or other CSS directives) may only extend selectors that appear within the same directive block. However, if the referenced selector by the @extend is also present in the outer scope, it fails applying the @extend rule and throws an error message as demonstarted by Example 2.42 and 2.43.

The raised error can be solved by creating a placeholder selector just like in the example above and reference it instead of the original selector. The newly created placeholder will not be present in the outer scope, thereby not leading to the same problem.

```
@media print {
  .selector { // this should be matched - it is in the same
      directive
    color: black;
  }
  .screenClass{
    @extend .selector; // extend inside media
  }
}
.selector { // ruleset on top of style sheet - extend should
    ignore it
  color: red;
}
```

Example 2.42: Sass: Extend inside directives

```
You may not @extend an outer selector from within @media.
You may only @extend selectors within the same directive.
From "@extend .selector" on line 6.
```

Example 2.43: Output: Extend inside directives

The error has been reported to the core contributors of Sass and can be found in the issue tracker of the repository[6].

Extend "all"

In Less, extend can take an optional parameter — the keyword "all". When it is applied, the extend modifies its behaviour and starts to act exactly like the @extend diretive of Sass by matching the selector as part of another selector as shown in Example 2.2.

Duplication detection

Sass implements the selector duplication detection feature. When merging selectors, *@extend* is smart enough to avoid unnecessary duplication, i.e. *.seriousError.seriousError* gets translated to *.seriousError*. In addition, it will not produce selectors that cannot match anything, like *#main#footer*. On the other hand, Less has no duplication detection implemented. Despite the possibly different CSS being generated because of the lack of duplication detection, the equivalence of the stylesheet generators is unaffected.

2.3.5 Mixins

There is a significant difference in the syntax of mixins as language constructs. In Less, all selectors are implicitly perceived as mixin definitions. On the contrary, Sass defines its own syntax for defining mixins.

---

[6]https://github.com/sass/sass/issues/2058

26

Another syntax-related difference can be found in the set of allowed characters regarding the mixin names. In Less, any selector can be used as the name for a mixin. However, Sass does not allow non-alphanumeric characters, except the _ (underscore), in its mixin identifiers. In addition, the identifier cannot start with a digit.

Scoping

The most significant semantic difference is in the scoping. In Less, the mixin definitions with the same identifier do not override each other. Actually, Less applies all the matching mixins of the caller scope. If none are defined, it will search the outer (parent) scopes until it finds at least one matching mixin and applies all the matching mixins of that scope.

However, Sass does not implement such a behaviour. We have identified two approaches in terms of finding a conversion method for this difference:

Fusion This approach merges all the matched mixin definitions into a single definition that will be applied upon calling the mixin.

Enumeration Enumeration converts each of the matched definitions into an individual mixin definition and instead of calling only one mixin, all the created mixins will be called in the order Less would execute them. However, the naming of the mixins should be reasonably resolved, i.e. by appending a serial number to them.

Both the aforementioned approaches will be used in order to not only provide an equal but also a readable and user-friendly Sass representation of the mixin features.

Access to variables   Another difference to be solved is the handling of locally undefined variables. If a variable is undefined in the local scope of the mixin, both stylesheet generators will start to look for the value in the parent scope moving towards the global execution context. If the value cannot be found, Sass throws an error announcing that the variable is not defined. However, Less does not stop there. It will also take a look at the caller scope.
To solve this difference, there are two cases that need to be further investigated:

1. The variable is defined in one of the parent scopes of the mixin declaration.

2. The variable is defined in the caller scope or one of its parent scopes.

In the first case, no action has to be taken, since both Less and Sass apply the same default behaviour and will take the value of the variable found in one of the parent scopes as shown in Example 2.44 and 2.45.
In the second case, the parameter list of the mixin should be extended by the locally undefined variable. When calling this mixin, the missing variable should be passed along with the other parameters as demonstrated in Example 2.46 and 2.47. This method will be used in conjunction either with **fusion** or **enumeration** whenever the conditions of this case are met.

```
@x: 1;
.mixin() {
  z-index: @x; // x = 1
}
#namespace {
  .mixin();
}
```

Example 2.44: Less: Parent scope variable

```
$x: 1;
@mixin __class__mixin() {
  z-index: $x;
}
#namespace {
  @include __class__mixin();
}
```

Example 2.45: Sass: Parent scope variable

```
.a(@a) {
  z-index: @x;
}
#namespace {
  @x: 1;
  .a("val");
}
```

Example 2.46: Less: Caller scope variable

```
@mixin __class__a($a, $x) {
  z-index: $x;
}
#namespace {
  $x: 1;
  @include __class__a("val",
      $x: $x);
}
```

Example 2.47: Sass: Caller scope variable

Selectors as mixins

In case of selectors as mixins, **fusion** will be applied. The name of the created mixin should be properly sanitized based on the regulations set by the parser of Sass. The conversion of this feature is shown in Example 2.48 and 2.49.

```
.a {
  // body
}
.a() {
  // different body
}
.class {
  .a()
}
```

Example 2.48: Less: Selector as mixin

```
@mixin __class__a {
  // body of .a selector
  // body of .a() mixin
}
.a {
  // body
}
.class {
  @include __class__a;
}
```

Example 2.49: Sass: Selector as mixin

28

```
#outer {
  .inner {
    color: red;
  }
}
.c {
  #outer > .inner;
}
```

Example 2.50: Less: Namespacing

```
@mixin __id__outer__class__inner {
  // the body of the namespaced mixin
}
#outer {
  .inner {
    //body
  }
}
.c {
  @include __id__outer__class__inner;
}
```

Example 2.51: Sass: Namespacing

Namespaces

Selectors in Less can behave like mixins and they can also be nested. Hence, Less offers a way to organize the defined mixins using the *namespacing* feature. Sass cannot organize its mixin definitions in such means. The solution is to copy the mixin definition and place it outside the namespace. The identifier of the copied mixin should reflect the namespacing structure as shown in Example 2.50 and 2.51.

Namespace and mixin guards

Less implements guards similar to CSS and its @media and @supports conditions instead of control directives as it is in case of Sass. The solution is to convert the guards to an equal representation of *@if statements*. The created control directive should be placed inside of the mixin definition encasing its content in both the namespace and mixin guard cases as demonstrated in Example 2.52 and 2.53. In case of namespace guards, all of the mixin definitions placed inside of the namespace should inherit the created control directive.

```
#namespace when (@mode=huge) {
  .mixin() {
    // body
  }
}
```

Example 2.52: Less: Namespace and mixin gurads

```
@mixin __id__namespace__class__mixin() {
  @if $mode == huge {
    // body
  }
}
```

Example 2.53: Sass: Namespace and mixin gurads

Important keyword

Less implements a shorthand for marking all property values defined in a particular mixin as `!important`. There is no language construct in Sass that would make this feature possible to reproduce. However, it can be solved by creating an extra parameter called `$__l2s__important` with a default value set to `null` as shown in Example 2.54 and 2.55. In order to keep the converted Sass code as clean as possible, this extra parameter should be added only to mixins that are called with the `!important` keyword.

```
.foo() {
  color: #fff;
}
.important {
  .foo() !important;
}
```

Example 2.54: Less: The !important keyword

```
@mixin class_foo ($__l2s__important: null) {
  color: #fff #{$__l2s__important};
}
.important {
  @include class_foo($__l2s__important: !important);
}
```

Example 2.55: Sass: The !important keyword

Parametric mixins

To create an equal representation of parametric mixins, **enumeration** will be used. Fusion cannot be applied in this case because the arguments can be set with default values. In addition, each mixin can specify a different value for the same argument. To resolve the naming of the parametric mixins, each of the converted mixin should be provided with a serial number appended to its name as seen in Example 2.57, which is the equal Sass representation of Example 2.56.

```
.mixin(@color; @value: uppercase) {
  // body
}
.mixin(@color; @value: 2; @margin: 2) {
  // body
}
.selector {
  .mixin(#008000);
}
```

Example 2.56: Less: Parametric mixins

```
@mixin __class__mixin__1($color, $value: uppercase) {
  // body
}
@mixin __class__mixin__2($color; $value: 2; $margin: 2) {
  // body
}
.selector {
  @include __class__mixin__1(#008000);
  @include __class__mixin__2(#008000);
}
```

Example 2.57: Sass: Parametric mixins

31

@arguments variable   The `@argument` variable has a special meaning inside Less mixins. It contains all the arguments passed to it, when it was called. Sass does not implement such a special variable, thus it needs to be created by the converter. It should create the variable `$arguments` that will have a simple list containing all the parameters specified in the mixin definition as shown in Example 2.58.

```
@mixin class_box-shadow($x: 0, $y: 0, $blur: 1px, $color: #000) {
  $arguments: ($x, $y, $blur, $color);
  // body
}
```

Example 2.58: Sass: @arguments variable

```
@mixin __class__mixin($switch, $color) {
  @if $switch == dark {
    color: darken($color, 10%);
  } @else if $switch == light {
    color: lighten($color, 10%);
  }

  display: block;
}

$switch: light;
.class {
  @include __class__mixin($switch, #888);
}
```

Example 2.59: Sass: Pattern matching

Pattern matching   In Less, different types of mixins exist and all possess a particular behaviour. One of those is pattern matching. This feature can be seen as an abstraction of a hypothetical switch directive demonstrated in Example 2.10. The converter should identify the mixins implementing this feature and apply the **fusion** approach to merge them into one mixin definition. This feature can be then converted into the *@if-@else if* Sass control directive as shown in Example 2.59. The body of the mixin definition that accepts a wildcard parameter should be placed outside of the created control directive, since it would be always called.

```less
.unlock(@value) { // outer mixin
  @value: 2 * @value;
  .doSomething() { // nested mixin
    declaration: @value;
  }
}
#namespace {
  .unlock(5); // unlock doSomething mixin
  .doSomething(); // mixin defined in doSomething became callable
}
```

Example 2.60: Less: Nested mixins

```scss
@mixin __class__unlock($value) {
  $value: 2 * $value;
  //body
}
@mixin __class__doSomething($value) {
  $value: 2 * $value;
  declaration: $value;
}
#namespace {
  @include __class__unlock(5);
  @include __class__doSomething(5);
}
```

Example 2.61: Sass: Nested mixins

Nested mixins

Nested mixins are syntactic constructs, that are not allowed in Sass. Those constructs should be detached and placed into the context of their parent mixin. The detached mixins can reference variables that are defined in their former parent mixin. In such case, those variables should be imported into the definition body of the detached mixin, as well. In addition, they should also inherit all the parameters of their parent mixin, so they could be called with the same parameters as shown in Example 2.60 and 2.61. That way can we ensure that the detached mixins will have access to the same context as if they were nested. However, the drawback of this method is the redundant code it creates, and thus breaking the DRY principle.

Mixins as functions

All mixins in Less possess a default behaviour. The variables and mixins defined inside of them are reachable from the caller scope. Due to this feature, it is necessary to modify the way the converter handles the scoping difference.

If a variable or a mixin is not defined in the caller scope, the converter should look first inside of the definition of all mixins being called in the caller scope before moving to the outer scopes. In case the variable definition is found in one of those mixins, a function directive should be created out of that definition and named after the missing variable. After that, the referenced variable should be replaced by a function call. The created function, as well as the mixins, can reference to variables defined in the caller scope. In such case the same principle should be applied as in Example 2.47. The parameter list of the function should be extended by the undefined variables, i.e. the variable @x in Example 2.62 and 2.63.

```less
.mixin() {
  @width:  100% / @x;
  // body
}
.caller {
  @x: 3;
  .mixin();
  width:  @width;
}
```

Example 2.62: Less: Mixins as functions

```scss
@mixin __class__mixin($x) {
  $width: 100% / $x;
  // body
}
@function width($x) {
  @return 100% / $x;
}
.caller {
  $x: 3;
  @include __class__mixin($x);
  width: width($x);
}
```

Example 2.63: Sass: Mixins as functions

Passing rulesets to mixins

Both in Less and Sass it is possible to pass rulesets to mixins. Less uses detached rulesets, whereas Sass uses content blocks. The main difference is, that Less is able to pass multiple rulesets to a mixin via parameters, since detached rulesets can be stored in variables as demonstrated in Example 2.64. On the other hand, Sass does not support rulesets as data type. It can pass only one content block to the mixin, which is accessed via a special directive inside of that mixin - the @content directive as shown in Example 2.65.

Another difference is, that while in Less it is possible to define mixins inside of a detached ruleset, Sass cannot do the same with its content block. In Less, this feature leads to a special behaviour of the detached rulesets. Both definition and caller scopes are available to them. If both scopes contain the same variable or mixin, the declaration scope value takes precedence. In addition, by including a detached ruleset into any other structure (e.g. another ruleset or mixin definition) all its properties (including the defined mixins and variables) will be accessible to the scope the detached ruleset had been included into. Using this feature, it is possible to dynamically call different implementations of the same mixin by calling the detached ruleset in different scopes. Sass cannot call mixins dynamically. Due to these characteristics of detached rulesets, we have not found an effective way — which would not break the DRY principles — to convert this feature into an equal Sass representation. After consultation with Red Hat, we decided to find an effective conversion method during the future work on the converter, since they do not use this feature of Less in their projects.

34

```
.apply-to-ie6-only(@content) {      @mixin apply-to-ie6-only {
    @content();                          @content;
}                                   }
.apply-to-ie6-only({                @include apply-to-ie6-only {
  #logo {                             #logo {
    color: #fff;                        color: #fff;
  }                                   }
});                                 }
```

Example 2.64: Less: Detached ruleset   Example 2.65: Sass: Passing content block

2.3.6   Import Directive

Both stylesheet generators implement the `@import` directive for importing other files that are either CSS or another dynamic stylesheet. However, there are certain restrictions in Sass regarding this feature.

The imported file will be compiled as CSS, if:

- its extension is *.css*,

- the filename begins with *http://*,

- the filename is a *url()*,

- the `@import` has any media queries.

If none of the above conditions are met and the extension is *.scss* or *.sass*, then the named Sass or SCSS file will be imported. If there is no extension, Sass will try to find a file with that name and one of the sass extensions and import it. If the extension is any other, than *.css*, *.scss* or *.sass*, Sass will not be able to read it, even if it contains valid Sass code. However, Less treats those files as if they were valid Less files and will try to compile them. To solve the difference, the converter will have to append a *.scss* or *.sass* extension to those filenames.

Import options

Less implements the following options that modify the behaviour of the @import directive:

- **reference** Compiles a Less file but does not include it into the final output.

- **inline** Includes the source file in the output without processing it.

- **less** Treats the import as a Less file regardless its extension.

- **css** Treats the import as a CSS file regardless its extension.

- **once** Includes the file only once (default behaviour).

- **multiple** Includes the file even if it had been previously imported.

35

- **optional** Continues in compilation if the file is not found.

More than one keyword per `@import` is allowed in Less. However, Sass natively supports only the options *once* and *reference*, where the former is the default behaviour, too. The latter option can be achieved by using Sass `partials`. All the other options cannot be reproduced in a Sass. However, the converter may manipulate the imported file extensions in order to achieve same results. The converter may even omit the `@import` directives containing the *optional* keyword if the file being imported cannot be found.

There is no known way to reproduce the funcionality of the *inline* option in Sass.

2.3.7 CSS Guards

Guards can be also applied to CSS selectors in Less. The Sass representation of these guards is an enclosing *@if* directive. However, if the guard is appended to a base-level rule containing the parent-selector-referencing character (`&`), Sass should omit that selector and enclose only its child rules as shown in Example 2.66 and 2.67.

```
@my-option: true;

& when (@my-option = true) {
  button {
    color: white;
  }
  a {
    color: blue;
  }
}
```

Example 2.66: Less: CSS guard

```
$my-option: true;

@if $my-option == true {
  button {
    color: white;
  }
  a {
    color: blue;
  }
}
```

Example 2.67: Sass: CSS guard

2.3.8 Loops

Loops in Less are achieved through recursively called guarded mixins as shown in Example 2.69. This behavior can be also reproduced in Sass using the *@if* control directive as demonstrated in Example 2.69.

```
.generate-columns(4);

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

Example 2.68: Less: Loops

```
@mixin generate-columns($n, $i: 1) {
  @if $i <= $n {
    .column-#{$i} {
      width: ($i * 100% / $n);
    }
    @include generate-columns($n, ($i + 1));
  }
}

@include generate-columns(4);
```

Example 2.69: Sass: Loops

2.3.9 Merge

This feature is unique to Less. I have not found a solution, that could express such a feature using the existing language constructs of Sass, despite of the fact that Sass possesses a wide range of built-in functions for list manipulation.

2.3.10 Parent Selector

Parent selector reference is available in both languages. They work absolutely the same way with one exception. Sass is unable to use multiple parent selector references joined by an empty string (&&) — shown in Example 2.70. The solution is to create a "placeholder variable", which stores the value of the parent selector and use its interpolated form instead. It can be seen in Example 2.71.

```
.link {
  & + & {
    color: red;
  }
  & & {
    color: green;
  }
  && {
    color: blue;
  }
  &, &ish {
    color: cyan;
  }
}
```

Example 2.70: Less: Parent selector

```
.link {
  $__l2s__parent: &;
  & + & {
    color: red;
  }
  & & {
    color: green;
  }
  &#{$__l2s__parent} {
    color: blue;
  }
  &, &ish {
    color: cyan;
  }
}
```

Example 2.71: Sass: Parent selector

2.3.11 JavaScript Evaluation

Sass is not able to evaluate JavaScript, at all. Hence, the JavaScript code has to be evaluated by the converter and the returned value must be output in the generated Sass code, instead. The equivalent Sass representation of Example 2.21 can be seen in Example 2.72.

```
$var: "HELLO WORLD!";
```

Example 2.72: Sass: Parent selector

2.4 Existing Conversion Possibilities

There are several existing transformation tools between Less and Sass already being used in practice. However, they all are based on a string replacement method using regular expressions which turned out to be imprecise and thus insufficient mainly because of the semantic differences and different language characteristics that we mentioned in the previous sections. We presented the methods to rewrite most of the Less language features into an equal Sass representation. We also learnt, that some of the features cannot be expressed using the Sass language constructs due to essential differences in the character between declarative and imperative programming paradigms and their conversion would be unefficient and breaking the DRY princpiles.

Using the string replacement conversion tools in many cases results in an incomplete and incompatible conversion that needs a lot of manual afterwork. These tools are great for small projects that use only the elementary features of Less. However, an approach based on *abstract syntax tree transformation* (henceforth, AST) yields a much more precise result according to our experience. Such tool could be used for converting larger projects that use several of the more advanced Less features.

In the following chapter we will describe the design and implementation of such a converter.

Chapter 3

Design and Implementation

This chapter describes the design and implementation of the converter proposed in this work. The goal of the converter is to transform dynamic stylesheets written in Less into their Sass representation. We have implemented the converter in Ruby and partly in JavaScript. Ruby has been choden for the following two reasons:

- It is a very expressive object-oriented programming language.

- Sass is also implemented in Ruby.

At the time of writing, Sass is already an eight years old project and throughout this time, it has been ported to several languages. For instance, the LibSass project implemented the Sass engine in C/C++ which makes the compilation a lot faster. However, the main development of the project still uses Ruby.

Less and Sass are the two most popular CSS preprocessors of our time with a lot of contributors. New features are coming on a regular basis. In order to stay up-to-date with both projects, thus lowering the maintainability costs, we have decided to use the engines of both Less and Sass heavily throughout the conversion process, even though it has a big impact on the performance of the converter.

The proposed conversion process is outlined in Figure 3.1 and can be devided into three major parts:

Parsing In this phase, the input Less code is being processed by the converter and is transformed into an abstract syntax tree by the Less engine.

Transformation During the transformation phase, the Less AST is modified and then converted into an equal Sass AST representation. Here, the Sass engine is used to create the AST representing the generated Sass code.

Code generation Finally, the Sass engine is used again to convert the generated AST into a human readable code, which is the final output of our converter.

In the following sections, we will describe the way each of the major parts is designed, the reasons behind this particular design was chosen as well as optimizations that can be applied to further improve the performance of the converter.
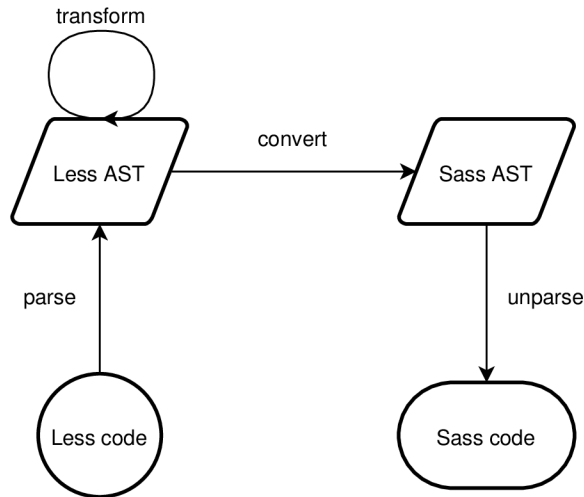
Figure 3.1: The conversion process

3.1 Parsing

The input of the parsing phase is a dynamic stylesheet written in Less, and the output is a Ruby object representing the AST of the input stylesheet. It is the most unefficient part of the conversion process. A part of the parser had to be implemented in JavaScript, since the original parser engine of Less.js is used to return the AST of the input. During this process, the input is processed seven times in total. In order to create a Ruby object representation of the AST, the following steps have to be taken:

1. A JavaScript program called *less_parser.js* has been written to parse the input Less code. This program serves as a wrapper around the parser of the Less engine. Since our converter is developed in Ruby, it is necessary to call the wrapper program through a system method call. The *Kernel* module of Ruby implements several methods (i.e. Kernel#system, Kernel#') for running commands in a subshell. We use its backticks method[1] to call Node.js — a JavaScript runtime built on Chrome's V8 Javascript engine — that runs our wrapper program as shown in Example 3.1.

2. The wrapper program calls the parser of the Less engine, which processes the input for the first time and returns an object representing the AST of the Less code.

3. The tree obtained in step 2 is further traversed by the visitor method of the wrapper program. This method appends a new property to each of the visited objects. The property stores the name of the prototype the object inherits from. It is a costly operation but needed. Otherwise, the converter written in Ruby would not have access to these information.

4. The modified tree from step 3 is converted into a JSON string then printed to the standard output and thus returned by the wrapper program.

5. The system call method reads the output data and stores it in the **string_ast** variable as shown in Example 3.1.

---

[1]

6. In order to be able to traverse the AST in our converter, the JSON string is necessary to be parsed back into a JSON object.

7. To be able to manipulate and transform the Less AST received from the input efficiently, the converter implements classes representing each node type of the AST[2]. The type of node is given by the added property in step 3.

   In this final step, the previously received JSON object is visited in a pre-order walk. During this process the aforementioned classes of the converter are used to recreate the nodes based on their types and thus transform the JSON object into a Ruby object representing the Less AST, which is returned by the parsing phase of the conversion.

```
string_ast = 'node path/to/less_parser.js path/to/input.less'
```

Example 3.1: Ruby: Using Node.js to run the wrapper program

The AST in its new form is easier to manipulate. The objects, it is composed of, are instantiated from classes that implement their own specific transformation and conversion methods.

As stated earlier, the parsing process is very inefficient due to multiple tree traversals and IO operations. To optimise this process, a custom parser implemented in Ruby is needed which would directly yield the required object representation of the Less AST, thus reducing the total number of tree traversals as well as the IO operations to one.

3.2   AST Transformation

The input of the transformation phase is an object representing the Less AST, and the output is an object representing the Sass AST. In both cases, the object references the root node of the tree.

During the transformation process, the Less AST is manipulated and converted into its Sass representation using the Sass engine. The Less and Sass engines both implement prototypes or classes, respectively, representing the nodes of their AST. However, they do not implement the same nodes. The language contructs of Sass are represented differently than the ones of Less. The difference between the representations is solved by the conversion methods implemented by the Less nodes of the converter. Those methods return the adequate Sass node representation, which is created using the Sass node classes implemented by the Sass engine[3].

Abstract syntax tree

Figure 3.2 shows an extract from the structure of the Less AST. There are 34 different node types and each of them are composed of other nodes. Literal nodes like *AnonymousNode,*

---

[2]https://github.com/less/less.js/tree/master/lib/less/tree
[3]https://github.com/sass/sass/tree/stable/lib/sass/tree

ColorNode and *UnitNode* are the leaves of the tree. The complete tree structure would be too immense for including it as a whole.

Each node of the tree represents a syntactic construct, e.g. ruleset, variable declaration, expresssion, unit value, etc. and has its own transformation method implemented based on its environment.
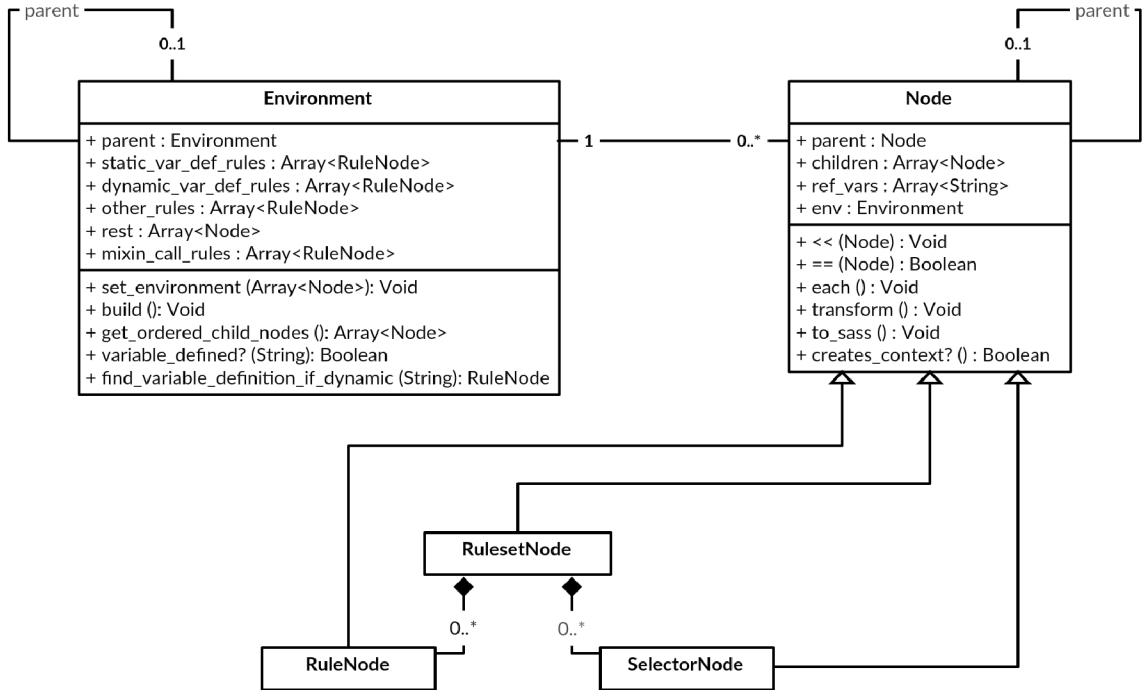

Figure 3.2: AST decomposition

Environment   In order to convert some of the Less language features, variable values and mixin parameters are needed to be evaluated. The implementation of environments and their ability to interpret Less code serve this purpose.

The environment represents the lexical scope of the given node and it is created in the transformation phase of the conversion process. Each created environment has a reference to its parent environment except the global one which sits on the bottom of the execution stack. A new environment is created whenever a node is traversed that represents a new scope, e.g. *RulesetNode, MixinDefinitionNode, DetachedRulesetNode*, etc.

Transformation process

The transformation can be described as a two step process:

1. A semantic preservation transformation is performed on top of the Less AST. It means that both the original and the transformed AST would yield an equal Less stylesheet. During that process, the order, naming and parameter list of some declarations may change without affecting the CSS output generated by the dynamic stylesheet. In this step, the execution context — also called environment — is set up. The process

traverses the whole AST in a pre-order walk and applies the respective transformation method on the nodes. A pre-order walk is applied because of the variable and parameter evaluation, which proceeds from the top of the execution stack to its bottom.

2. The altered AST is traversed one more time and the nodes are converted into their Sass representation. However, the traversal is performed in a post-order walk this time, so the Sass AST gets built from the bottom to the top, since the Sass nodes must be instantiated using their child nodes as parameters.

3.3 Code Generation

The input of the code generation phase is a Sass AST, and the output is one or more Sass dynamic stylesheets.

Sass supports two syntaxes — *SCSS* and *Sass* — and implements a converter between them. In this phase, we use the code generation capabilities of the Sass engine and its methods that generate Sass code in the respective syntaxes from the input AST. The optional argument *--TARGET_SYNTAX* can be specified in order to tell the converter, which of the possible syntaxes should be applied in the code generation process. The default syntax is SCSS.

Less stylesheets can reference multiple other stylesheets by importing them and so does Sass. In the code generation process, all the included stylesheets will be output, as well. The Sass AST is recursively traversed in a pre-order walk. The code generation process consists of the following steps:

1. The tree is traversed node by node until an *ImportNode* is found or the last leaf node is reached.

2. If an *ImportNode* is found, the code generation method is also called on the AST of the stylesheet it references. After that, the tree traversal continues as stated in step 1.

3. When the end of the tree is reached, Sass code is generated from the AST according to the selected syntax.

3.4 Supported Features

The Less to Sass converter, in its current state, is able to convert the following syntactic constructs and language features:

- selectors,

- CSS properties,

- CSS values including variables,

- variable interpolations,

- variable definitions,

- lazy loading.

43

Lazy loading

The lazy loading feature of Less is solved by importing the affected variable definitions according to Example 2.35. In the next step, the child nodes of each environment are put into a specific order:

1. static variable definitions, i.e. `$a: 100%;`

2. dynamic variable definitions, i.e. `$var: $a;`

3. all the rest, i.e. `@media`, `@import` directives,

4. rulesets (selectors with their declarations, mixin definitions),
   i.e. `.a, .b { color: red; }`,

5. mixin calls, i.e. `.lazy-eval-scope();`.

Using this specific order, we can ensure that variables and mixins are not referenced neither called before their definition.

Chapter 4

Testing

In this chapter, we describe the way we tested our converter. We first describe the process of testing we adopted. Then, we introduce the tool we created for our testing purposes and describe its design and implementation. Finally, in Section 4.3, we show the applied test suite and discuss the results of the test.

4.1   Testing Process

In order to verify the results of the converter, we compare the generated Sass stylesheet with the input Less code according to Definition 2.1. Our testing process thus consists of three steps described in Figure 4.1.

1. The input is converted into a Sass stylesheet.

2. Both the input and the generated Sass code are compiled, and each of them returns a CSS stylesheet.

3. These stylesheets are compared using a suitable CSS comparing tool, which returns `true` or `false`.

There are several possibilities to compare CSS stylesheets with each other. Some of the existing tools are CSS Comparer by Alan Hart[1] and CSSCompare by Bert Johnson[2]. Only the latter example is an open source project, thus we could not examine the source code of both projects. However, based on our experience, they both share the same principle. They use string-based comparison, which is quite efficient in combination with some CSS postprocessor, like PostCSS[3].

CSS postprocessors are able to normalize CSS files by eliminating the possible redundancy in selectors, sort these selectors alphabetically and also compress the files by removing unnecessary whitespaces. However, this kind of solution is not reliable, since CSS stylesheets with different design can lead to equivalent HTML output. For instance, Example 4.2 is equal to that of Example 4.1 because the second compound selector overrides the first selector definition and causes `.a` to receive the green color. Also, both specified paths

---

[1] http://www.alanhart.co.uk/tools/compare-css.php
[2] https://github.com/bertjohnson/CSSCompare
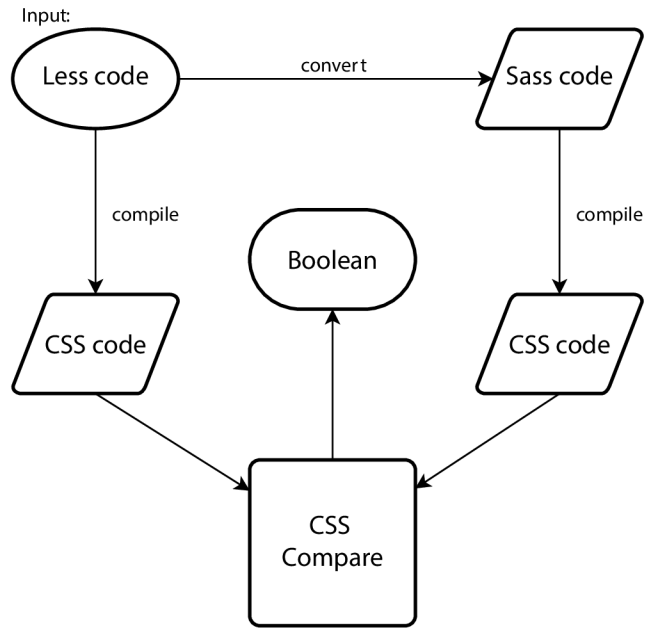[3] https://github.com/postcss/postcss

Figure 4.1: Testing procedure

are relative and would be evaluated the same way. The quotes used to wrap the paths do not affect the evaluation, either. When declaring the font-family CSS property, the use of wrapping quotes is also optional. However, these CSS stylesheets would not be evaluated as equal using the aforementioned tools.

```
.a, .b {
  color: green;
  background-image: url("path/to/file.jpg");
  font-family: Times New Roman;
}
```

Example 4.1: CSS: Simple example

```
.a { color: red; }
.a, .b {
  color: green;
  background-image: url('./path/to/file.jpg');
  font-family: "Times New Roman";
}
```

Example 4.2: CSS: Overriding example

Another possibility is to use *regression testing* that compares two CSS stylesheets graph-

46

ically. The CSS files are applied on the same HTML template, which is then rendered in the same arbitrary browser. After that, a screenshot is taken of both rendered pages. The images are compared pixel by pixel with a certain tolerance. This method is used at Red Hat, the proposing company of this thesis. According to its employees, this method is very slow and unreliable. In our opinion, this kind of testing is inaccurate and could lead to false results due to the *@media* CSS directives. Multiple screenshots may be taken at different screen sizes in order to reliably test the @media directives specified in the CSS stylesheets when using this testing procedure. However, the duration of testing would increase in proportion to the number of different conditions specified in the given CSS stylesheets.

In summary, our research regarding the CSS comparison tools showed that the existing solutions are either unreliable or too slow. This led us to the conclusion that a custom tool should be built as a part of the contribution of this thesis that is independent of HTML templates and is able to normalize not only the CSS selectors but also the values. In order to do so, it is necessary to define what does equality between two CSS stylesheets mean.

Definition 4.1. Two CSS stylesheets are equal if and only if their evaluation by the same arbitrary interpreter returns identical results.

Below, we describe our own approach and tool called CSS Compare for comparing CSS stylesheets according to Definition 4.1.

4.2 CSS Compare

CSS Compare is our AST-based comparison engine that is able to interpret CSS stylesheets according to the W3 specifications[4] and compare the results. We have built it for testing purposes of the Less to Sass converter. It is an open source project hosted on GitHub[5] and written in Ruby. It is also published on RubyGems.org, the gem hosting service of the Ruby community and can be accessed via the address https://rubygems.org/gems/css_compare.

We have decided to use the Sass engine for parsing and comparing purposes, since any valid CSS code can be also interpreted as valid SCSS code. Hence, the Sass parser engine is a suitable fit for the comparison tool, which works in the following way:

1. The tool receives two CSS stylesheets as input.

2. Both stylesheets are parsed and their abstract syntax tree is created.

3. These trees are traversed and interpreted. The product of the interpretation is the inner representation of the stylesheets in the form of a hash.

4. Finally, the hashes get compared.

4.2.1 The Inner Representation of Stylesheets

A CSS stylesheet defines the look and position of DOM objects in the browser under different @media and @support conditions. Based on these conditions, the adequate property value is applied to the DOM objects matching the specified CSS selectors.

---

[4]https://www.w3.org/Style/CSS/current-work
[5]https://github.com/vecerek/css_compare

47

The process of CSS interpretation creates the inner representation of CSS stylesheets according to the W3C specifications. Basically, it is a hash of selectors, which are defined in the CSS stylesheet. Each selector is represented as a key-value pair. The key is the name of the selector, whereas the value is another hash containing the properties declared under the particular selector. Each property is also represented in the form of key-value pairs. The key is the name of the property, whereas the value is another hash containing the defined @media conditions and the value of the particular property under the given condition. If the same property is declared multiple times under the same selector and @media condition, its value is overwritten by the last declaration in the hash of @media conditions and property values. The inner representation of the CSS code specified in Example 4.3 is shown in Example 4.4 in form of JSON.

```
.a > .b:first-child { color: red; }
@media (min-width: 768px) {
  .a > .b:first-child { color: green; }
}
```

Example 4.3: CSS: Inner representation

```
{
  ".a > .b:first-child": {
    "color": {
      "all": "red",
      "(min-width: 768px)": "green"
    }
  }
}
```

Example 4.4: JSON: Inner representation

In the interest of simplicity, the CSS values (`red` and `green`) demonstrated in Example 4.4 are output in the form they have been specified in the CSS stylesheet shown in Example 4.3. However, CSS Compare stores the property values as objects enclosing the Sass nodes obtained during the parsing phase of the comparison. The advantage of CSS values being stored as nodes over strings is that the operation of equation can be redefined. That way, the CSS values can be compared in their normalized forms.

The CSS values demonstrated in Example 4.5 are equal according to the W3C specifications. Firstly, both the specified paths in the `url()` function are evaluated relatively to the location of the CSS file declaring these values. Secondly, the use of single and double quotes is interchangable — in case the leading and trailing quotes are of the same type. Furthermore, when declaring font-family names, the use of quotes is optional. Finally, the use of extra whitespaces does not change the way the values are evaluated.

```
url("./path/to/file.jpg")    ==   url('path/to/file.jpg')
"Times New Roman"            ==   Times New Roman
rgb(0,0,0)                   ==   rgb(0, 0, 0)
```

Example 4.5: CSS: Value normalization

4.2.2 Selector Normalization

In order to build an exact comparison tool, CSS selectors have to be normalized, as well. The selectors specified in Example 4.6 refer to the same group of DOM objects.

```
.a.a                 == .a
.link:visited:hover  == .link:hover:visited
*:hover              == :hover
```

Example 4.6: CSS: Selector normalization

To normalize the CSS selectors, the following two operations have to applied:

- The redundancy in selector sequences should be removed. A selector sequence is a list of selector elements separated by an empty string.

- The elements of the selector sequences should be put in a defined order. That order has been assambled as follows:

  1. Universal selector (*).

  2. Element selector, i.e. *div*.

  3. Id selector, i.e. *#id*.

  4. Class selector, i.e. *.class*.

  5. Pseudo-selector, i.e. *:first-child*.

  Attribute selectors like *[type="text"]* are always tied to the preceding selector element. In addition, the selector elements should be also put in an alphabetical order within the same group of elements.

4.2.3 Supported CSS Features

CSS compare supports the comparison of the following list of CSS features:

- CSS selectors,

- color values[6] (e.g. `#f00` == `rgb(255,0,0)` == `red` == `cmyk(0,100,100,0)`),

---

[6]For color comparison, we use the Ruby gem called *color*: https://github.com/halostatue/color

- @media directive (also nested),

- @import directive (also conditioned),

- @font-face directive,

- @namespace directive,

- @charset directive,

- @keyframes directive,

- @supports directive,

- @page directive.

To further enhance CSS Compare, there are several improvements that can be implemented.

@media and @supports condition evaluation

In the current state of the project, these conditions are stored as strings. However, they should be evaluated and normalized similar to selectors.

Shorthand CSS property decomposition

CSS defines properties as shorthands for several other properties. For instance, the `border` property is a shorhand for `border-width`, `border-style` and `border-color` properties. To evaluate the CSS selector in an exact manner, these shorthand properties should be decomposed and stored in their basic form.

4.3   Test Suite

Our converter has not yet been applied to any major nor popular CSS framework implemented in Less. The test suit consists of short Less files (henceforth, examples), each demonstrating the use of a single language feature. We have written these examples in order to test the specific conversion methods of the supported language features. Table 4.1 shows the results of these tests and the size of the files. It also describes the performance of the comparison as well as the speed at which each phase of the conversion process is executed.

The examples can be divided into four categories based on the kind of the language features they contain:

- **Comments** Examples 1 and 2 contain block and line comments, respectively.

- **CSS features** In examples 3 to 11, CSS features like color values, dimensions and selectors are being tested. Example 11 contains the CSS properties enlisted in the CSS reference[7] of W3Schools, which is tested regularly with all major browsers.

- **Variable interpolation** Examples 12 to 18 contain several examples of variable interpolations in various contexts, i.e. property names, selectors and strings.

---

[7] http://www.w3schools.com/cssref/

- **Variables and values** In examples 19 to 23 variable definitions with various values (i.e. single quoted, double quoted strings and various color values) are being tested.

The performance of the converter and its parts has been measured using the Benchmark[8] module of Ruby. All tests have been run 100 times and the average values are being shown in the table. The received data confirmed our assumption regarding the inefficiency of the parsing phase of the conversion process, which has been discussed in Section 3.1.

#	Less file	(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	block_comment.less	P	4	272.09	127.20	0.13	0.03	0.01
2	line_comment.less	P	2	268.91	127.06	0.12	0.03	0.02
3	color_values.less	P	7	289.36	134.54	0.29	0.11	0.32
4	dimension_values.less	P	6	292.48	133.88	0.25	0.08	0.29
5	empty_selector.less	P	2	277.02	136.17	0.26	0.05	0.01
6	font_face.less	P	5	286.49	134.27	0.19	0.06	0.08
7	font_values.less	P	5	318.50	134.28	0.25	0.06	0.13
8	gradient_functions.less	P	4	319.81	139.15	0.30	0.11	0.20
9	keyframes.less	P	5	303.65	133.29	0.24	0.09	0.07
10	transition_values.less	P	4	299.13	143.01	0.30	0.05	0.12
11	properties_by_type.less	P	240	411.82	186.61	4.92	1.75	6.86
12	lazy_loading.less	P	11	303.30	140.10	0.31	0.08	0.20
13	lazy_loading_2.less	P	12	307.84	141.84	0.46	0.09	0.13
14	lazy_loading_3.less	P	15	311.50	138.06	0.69	0.12	0.13
15	multiple_interpolations.less	P	6	308.60	135.88	0.29	0.10	0.09
16	properties.less	P	7	319.59	143.72	0.29	0.07	0.24
17	selectors.less	P	6	290.65	133.16	0.19	0.06	0.16
18	selectors_expanded.less	P	6	284.45	136.02	0.17	0.07	0.15
19	urls.less	P	6	289.20	132.88	0.32	0.08	0.12
20	basic.less	P	6	285.12	133.37	0.23	0.05	0.17
21	double_quoted_string.less	P	6	284.89	140.72	0.20	0.05	0.12
22	single_quoted_string.less	P	6	289.67	132.72	0.20	0.06	0.11
23	unquoted_string.less	P	6	287.07	132.47	0.23	0.06	0.10
24	values.less	P	15	290.62	139.17	0.59	0.20	0.02

Table 4.1: Test Suite Performance

(1)	Passed/Failed
(2)	Number of lines
(3)	Total test time [ms]
(4)	Parsing time [ms]
(5)	Transformation time [ms]
(6)	Code generation time [ms]
(7)	Comparison time [ms]

---

[8] http://ruby-doc.org/stdlib-1.9.3/libdoc/benchmark/rdoc/Benchmark.html

Chapter 5

Conclusion

The goal of this work was to study the differences between the two most popular CSS preprocessors of our time, namely Less and Sass, and implement a converter between them. During the work, all of the differences have been explored, and several ways have been found to transform specific language constructs and features from one dynamic stylesheet language to the other.

It was shown that all of the language features can be reproduced in the target dynamic stylesheet language. However, the Less feature called detached ruleset — discussed in Section 2.1.3 and Section 2.3.5 under the pararaphs *Passing rulesets to mixins* — would be too costly to transform into its Sass representation as well as impractical due to breaking the DRY principle that is followed by both of the CSS preprocessor languages. After consultation with Red Hat, the proposing company of this thesis, we have decided to find an efficient conversion approach during the future work on the project. The converter is needed for the project called Patternfly[1], which is built on top of the popular CSS framework called Bootstrap[2]. Currently, neither of the frameworks use the aforementioned feature of Less.

To be able to verify the results of the converter, a custom AST-based comparison tool has also been built. This tool might be a more reliable and faster alternative to the presented existing solutions to comparing CSS stylesheets, which are based on either string comparison or graphical comparison methods. Our CSS comparison tool itself is a great benefit to developers who convert or optimize dynamic stylesheets on a regular basis and struggle with the verification of their work.

5.1 Discussion of the Achieved Results

The existing converters are based on string replacement methods using regular expressions, which cannot lead to full value conversions due to the different nature of declarative and imperative programming languages — Less and Sass, respectively. The converter described in this work supports the conversion of language features like lazy loading that cannot be adequatly converted by the existing solutions to the dynamic stylesheet conversion. However, the full support of some major features (i.e. mixins and extend) discussed in Section 2.1 is still missing. Therefore, our converter cannot be successfully applied on the aforementioned CSS frameworks.

---

[1] https://github.com/patternfly/patternfly
[2] https://github.com/twbs/bootstrap

Until the work on the remaining unsupported language features is finished, developers can use the conversion methods introduced in Chapter 2.3 as a guide to manually convert those features.

5.2   Future Directions

In order to eliminate the manual afterwork that follows the conversion of Less stylesheets to their Sass representation, the conversion methods of the currently unsupported language features need to be implemented.

Also, there are several opportunities to improve the performance of the converter. The most significant improvement could be reached by implementing a custom Less parser, which could lead to a five times faster parsing process of the converter compared to the current solution.

The converter can be further used to implement Ruby on Rails plugins that automate the conversion of Less stylesheets in web development projects using tools like Bower.js and Rails Assets.

The knowledge gained during the research of language differences between Less and Sass may greatly support the development of a bidirectional converter.

Bibliography

[1] Getting started: An overview of less, how to download and use, examples and more. „http://lesscss.org/".

[2] Less: Language features. „http://lesscss.org/features/".

[3] Sass documentation. „http://sass-lang.com/documentation/file.SASS_REFERENCE.html".

[4] Sass: Libsass. „http://sass-lang.com/libsass".

[5] Hampton Catlin, Natalie Weizenbaum, and Norman Clarke. About haml. „http://haml.info/about.html".

[6] World Wide Web Consortium. Style activity statement. „https://www.w3.org/Style/Activity", 2014.

[7] Matthew Dean. Less: The world's most misunderstood css pre-processor. „https://getcrunch.co/2015/10/08/less-the-worlds-most-misunderstood-css-pre-processor/", October 2015.

[8] Jacob Gube. 6 current options for css preprocessors. „http://www.sitepoint.com/6-current-options-css-preprocessors/", November 2014.

[9] Davood Mazinanian and Nikolaos Tsantalis. An empirical study on the use of css preprocessors. Technical report, Department of Computer Science and Software Engineering Concordia University, Montreal, Canada, 2016.

Appendices

List of Appendices

Appendix A

CD Content

/css_compare/ The directory contains the source code of CSS Compare v0.2.0. It has been released under the MIT license and can be also accessed at the URL address https://github.com/vecerek/css_compare/tree/v0.2.0.

/doc/ The directory contains the source files of the technical report written in LATEX. The files can be compiled using Makefile.

/examples/ A directory containing several Less files that can be converted to Sass stylesheets using the less2sass converter.

/less2sass/ The directory contains the source code of Less2Sass v0.1.1. It has been released under the MIT license and can be also accessed at the URL address https://github.com/vecerek/less2sass/tree/v0.1.1.

/README The manual containing further information regarding the installation and testing.

/xvecer17_technical_report.pdf The PDF file of this technical report.