Sless

Less Css through Scala

Bob Reynders & Thomas Van Strydonck

Introduction

Very importantly, all questions go onto the toledo forum, not into our mailbox. You are free to discuss problems with your fellow students on the toledo forum as long as you're not giving away solutions. Do not just dump parts of your own code.

The assignment is incremental, consisting of a compulsory base project and different optional extensions. In addition to implementing certain features, you will have to comply with certain code quality requirements. The project will be graded both automatically and manually. We will manually look at your code and deduct points if you makhttps://developer.mozilla.org/en-US/docs/Web/CSS/:nth-childe that task more difficult than needed, i.e., breaking software engineering principles you've been taught before such as writing 1k line functions or implementing your entire project in a single god class. Another aspect of this code quality requirement is that we do *not* want you to copy your code to solve each individual extension: we expect you to deliver a single project that works for each increment you chose to solve.

A flawless implementation of the base assignment gets you a passing grade. Your grades go up as you implement other features. Certain features impact the difficulty of others. Read the entire assignment and think before you act. Naively implementing the base makes the implementation of other features a nightmare.

If you feel a bit overwhelmed by Scala features, we personally write our projects with the following guidelines in mind:

- Use OO to encapsulate implementation details
- Use FP for small abstractions (e.g., collection API methods, higher-order functions)
- Favor immutability (the entire project can be written without mutable variables)

The estimated time you should spend on this project is around 40 hours. You should be able to complete at least the base project in this time span.

Submission Details

Before you submit check that you:

- completed the FILLME.md file with details about your project
- zipped your project so that it contains everything (including your tests)
- did *not* modify any provided `*.DSL`-file (except if you implemented extensions for Linting, see below)

High-level Description

In this project you will develop a DSL (Domain Specific Language) in Scala that compiles to CSS. However, since CSS is sorely lacking in language features, we mimic the approach of multiple DSL-languages before us (e.g. Less and Sass) and enhance CSS with our own features. In the end, your DSL, Sless (Scala-Less), will allow you to easily write down and manipulate ASTs (Abstract Syntax Trees) representing (an extended version of) CSS code and will allow transforming these ASTs into CSS-specification-compliant `Strings`, i.e. compiling our ASTs to CSS. This assignment is not entirely self-contained: it explains all basic terminology we use in the code, but expect to have to read up on Scala-, CSS- and Sass\Less-related topics online!

Getting Started

Open up your IntelliJ and select 'Import Project'. Select the `build.sbt` file to do an SBT-driven import. The `code\src` folder contains `main` and `test` subfolders, meant for your implementation of the project and the linking to our test suites (and your self-written tests later on), respectively. For each extension we have you implement, three separate files are present (this differs for the base project Sless-Base, but will be detailed below):

1. A `*DSL`-file in the `main\dsl`-folder, defining the trait you should implement for a particular extension. These `*DSL`-files also define shorthand notations for concepts from your Sless ASTs through the use of implicit classes. These shorthands will allow much nicer formatting of tests and input programs, making Sless a true DSL.
2. An `*Implementation`-file in the `test`-folder, where your implementations for a particular extension should be linked with the applicable DSL's.
3. A `*Test`-file in the `test`-folder, where we defined a few example tests for a particular extension to show off the Sless syntax and DSL API, and where you can append your own tests, should you wish to do so.

The `main\ast`-folder is where we expect you to implement your solution itself, i.e., the structure of your Sless ASTs (linking to the test suites and testing still happens in the `test`-folder, obviously).

Sless: Base Project

This section first describes the base project, the perfect implementation of which will grant you a passing grade, and then details the possible extensions to Sless-Base you can choose to implement. To reiterate, the base project is compulsory, the extensions optional. Every content section of the assignment will contain a listing of all relevant files. Additionally, keep in mind that some extensions will force you to backtrack and change implementations of files from previous extensions (the worse your solution's design is, the more severe this backtracking will become). Write and include your own tests to convince both yourself and us that your implementations are correct! (We will run our own additional tests as well)

The base project consists of a CSS implementation with basic Less features and grants you a passing grade upon flawless completion. The functionality to be implemented in the base project consists of the following five points (detailed in the next five subsections, in order):

1. Implement all DSL traits for the base project, i.e. implement Sless ASTs and link your implementation to our DSL surface syntax
2. Generate valid CSS code (in `String` format) without whitespaces from Sless ASTs, i.e. implement no-space printing to CSS
3. Generate pretty-printed, valid CSS code (in 'String** format** with customizable indentation from Sless ASTs
4. Demonstrate the power of a Scala CSS DSL by implementing certain scenarios that show how some Less features are automatically available
5. Efficiently implement a couple of lint tool-style checks on your Sless ASTs

1. Implementing all DSL traits for the base project

DSLs: `BaseDSL`, `PropertyDSL`, `SelectorDSL`,`ValueDSL`, `Compilable`

Tests: `CssImplementation`, `CssTest`

This section further details the structure of the Sless-Base code. By the end of this section, you should be able to write your own implementations, in the `ast`-folder, for the DSL files listed above and link your ASTs by implementing the DSLs mentioned above in the `CssImplementation` file. We first introduce some necessary CSS-concepts and illustrate the goal of the project more concretely by means of an introductory example.

We start of by discussing the constituents of a CSS-sheet and their proper nomenclature. In case of CSS-related ambiguity or incomplete information, consult the links we interspersed throughout our explanation below and consult a CSS reference manual (we used this one). In the rest of the assignment, the tt-font words between brackets describe the names we use for concepts in the assignment's code.

A CSS sheet consists of a sequence of style rules (`Rule` in the code), that each specify the style that should be applied to a certain collection of HTML elements. The collection that the rule

should be applied to is specified by its selector (`Selector`), and the effect of a selector on these elements is specified by a semicolon-separated sequence of declarations (`Declaration`). A single declaration consists of an assignment of a value (`Value`) to a property (`Property`).

Consider the following CSS-sheet as an example:

```
strong {
  color: red;
}

div.menu-bar li:hover > ul {
  display: block;
}
```

This sheet has two style rules, with selectors `strong` and `div.menu-bar li:hover > ul` respectively, both containing a single declaration, `color: red` and `display: block` respectively, where the properties are `color` and `display`, and the values are `red` and `block`.

In this section, we want to be able to write down similar (and in future extensions, more advanced) sheets in our own Sless surface syntax. To this end, we have defined the DSL-files (and homonymous traits) `BaseDSL`, `PropertyDSL`, `SelectorDSL` and `ValueDSL` for you to implement. The same example sheet in the Sless-Base syntax defined in these DSL files looks as follows in Scala:

```
val strong = tipe("strong")
val color = prop("color")
val red = value("red")

val div = tipe("div")
val li = tipe("li")
val ul = tipe("ul")
val display = prop("display")
val block = value("block")

css(
  strong {
    color := red
  } ,
  (div.c("menu-bar") |- (li :| "hover") |> ul) {
    display := block
  }
)
```

We now go over each file/trait in order, thereby clarifying the above notation. Consult the source code and the previously mentioned sources of documentation for more detailed information.

The trait `BaseDSL` allows you to specify the specific type *you* use for the implementation of CSS sheets, i.e. the types `Rule`, `Selector`, `Declaration`, `Selector`, `Declaration`, `Property` and `Value` mentioned above. The specific type you use might be subject to change as you implement more extensions. There is also a type `Css` representing your type for entire style sheets. The method `css` allows us to construct a CSS sheet, as shown in the example above, and uses the internal method `fromRules`.

The trait `ValueDSL` is next to trivial and allows for the construction of `Values` from `Strings` through the `value` method.

The trait `PropertyDSL` does the analogue for `Propertys` using the `prop` method. It also allows for the construction of a `Declaration` from a `Property` and a `Value` using the `assign` method, and introduces the shorthand `:=` to do this.

The trait `SelectorDSL` will require the bulk of your implementation effort for this section. It defines methods that allow constructing all different shapes of `Selector` that we will need for our base implementation. The base selectors can be subdivided into selector modifiers (taking a selector as an argument and appending something), selector combinators (making two existing selectors into a new one) and selector constructors (creating selectors or sequences of selectors). The selector modifiers consist of class selectors (`className`), ID selectors (`id`), attribute selectors (`attribute`, where we currently only allow exact matching `=` on values, e.g. no `~=` operator), pseudo-class selectors (`pseudoClass`) and pseudo-element selectors (`pseudoElement`). The selector combinators consist of the descendant combinator (`descendant`), the child combinator (`child`), the adjacent sibling combinator (`adjacent`) and the general sibling combinator (`general`). The selector constructors consist of the selector list creator (`N` with its internal representation `group`), the type selector creator (`tipe`, since `type` is a keywoard) and the universal selector (`All`). Read up on any type of selector you did not yet know online. Finally, the `bindTo` method allows taking a selector and a sequence of declarations, and making these into a style rule `Rule`. The `SelectorShorthand` implicit class contains useful shorthands for most variants of the above. Check and see if you are now able to understand all of the notation in the previous example.

We did not yet mention the purpose of the `Compilable` trait. It contains two methods `compile` and `pretty`, responsible for respectively spaceless printing and pretty printing, that will be specified in more detail in the following two subsections.

Once you have implemented all of the above (which includes the next two subsections), you can link your implementation to our test suite in `CssImplementation`, by replacing in the `???` value. If this typechecks and builds, try running the tests in `CssTest`.

NOTE: since the above example CSS does not contain any reuse of code and the corresponding Sless sheet does not leverage any advanced DSL features, the result might seem unwieldy at first glance. This will change, however, as you add more Sless features, and as style sheets grow more complex. For more examples of Sless syntax use, just consult the `*Test`-file for the extension you are interested in.

2. No-space printing

DSLs: `Compilable`

Tests: `CssImplementation`, `CssTest`

This section details how to implement the `compile` method for the `Combilable` trait. This method accepts a Sless AST and returns a valid CSS source file in `String` format. This string should *not* contain *any* whitespaces or newlines! The only exception to this rule is that you should introduce a space if not including a space would cause the CSS program's meaning to be altered, e.g. introduce a space between the type selectors `id1` and `id2` under the descendant selector combinator, to avoid getting a different type selector `id1id2`. This compilation function is central to the usefulness of our DSL, and also to any test cases we wrote or you will write in the project extensions. Expect the implementation of this method to be updated with nearly every extension you implement and prepare for it accordingly now. Consult the CSS standards referenced above in case you are unsure how to print a specific CSS element.

3. Pretty printing

DSLs: `Compilable`

Tests: `CssImplementation`, `CssTest`

This section details how to implement the `pretty` method for the `Combilable` trait. This method accepts a Sless AST and returns a valid, pretty printed CSS source file in `String` format. This string *should* contain whitespaces and newlines in the correct places! Concretely, the `spaces` parameter specifies the number of spaces that each `Declaration` should be indented with. Additionally, a space should be added after the colon that separates a property and its value, after a selector but before the brace starting the block of declarations, after each comma in a list selector and before and after each selector combinator symbol (except for the descendant combinator, for obvious reasons). A newline should be added after the opening brace of a declaration block, after each declaration in such block, and two newlines should be added after every declaration block's closing brace (except for the last closing brace). As with the regular printing method, expect the implementation of this method to be updated with nearly every extension you implement and prepare for it accordingly now.

Once you finish this and the previous section's implementation, run the tests in `CssTest` to check your implementation, and include your own in case of doubt.

4. Some Less features are automatically available

Tests: `LessVariableImplementation`, `LessVariableTest`

In this section, we explore the convenience of using Scala as a host language for our DSL. Because we use Scala, there is some Less functionality we get for free, without having to extend our

AST implementation. Here, we explore the use of variables (cfr. variables in Less). Implement `LessVariableImplementation` by following the instructions there, and run `LessVariableTest` to partially check your implementation.

NOTE: Due to an IntelliJ bug, implementing the definitions might result in a false positive typing error during implementation in some versions. Build your project to be sure the error is actually there. This same type of mirage error pops up from time to time further on in the assignment as well, so perform a quick check before you scurry to the toledo forums :)

5. Lint tool-style checks

DSLs: `LintDSL`

Tests: `LessLintImplementation`, `LessLintTest`

In this section, we develop a very basic lint tool to perform checks on Sless ASTs. Lint tools or linters are tools that analyze source code (Sless ASTs in our case) to find different types of stylistic and other programming errors, as well as detect code smells (see e.g. Wikipedia). Since the static guarantees that the CSS language offers the programmer are so minimal, it makes sense to have your CSS code analyzed by an external lint tool before putting it online. A whole range of CSS validation tools has been developed for this purpose, one of which is CSS Lint. CSS Lint is rule-based and performs a sequence of boolean checks on the CSS input; we will implement a couple similar rules in this section. Implement the methods specified in `LintDSL` to create your lint implementation, connect to our tests in `LessLintImplementation` and test using `LessLintTest`. Create your own tests if you deem this necessary.

Sless-Extensions

This section lists possible extensions to the base project you can implement to increase your grade. Do not bother implementing any of these extensions if you do not yet have a fully functioning base project! The list of extensions below is *non-chronological*, i.e. we do not expect you to implement these extensions in order. Neither do we expect you to implement each and every one of the extensions. Just pick a couple you like and work on those.

Comments

DSLs: `CommentDSL`

Tests: `CommentImplementation`, `CommentTest`

Extend your Sless functionality with comments. We only support adding comments to rules and declarations. They have to be preserved both in the spaceless compilation result as well as in the pretty printed result. In both cases the form they take is `/* comment */` given a `String comment`. Look at the tests for an example.

Merging CSS sheets (imports)

DSLs: `MergeDSL`

Tests: `MergeImplementation`, `MergeTest`

Extend your CSS sheets with the functionality to merge multiple sheets into one (thereby simulating a primitive import system). Concretely, what you have to do is check for identical selectors (this also includes elements of selector lists and nested selectors, if you implemented those) between all sheets, append all declarations for unique properties to the rightmost selector (if they weren't part of the rightmost selector yet), and for non-unique properties only append the rightmost declaration to the rightmost selector (if it wasn't part of the rightmost selector yet). Finally, you only retain the rightmost selector and erase the others (erasing any rules without selectors in the process). Check the `MergeTest`-file for more details, and implement the `MergeDSL` in `MergeImplementation`.

Mixins

DSLs: `MixinDSL`

Tests: `MixinImplementation`, `MixinTest`

Similarly to how we noticed that Sless-Base automatically provided us with a variable feature without requiring extra effort, this section observes similar advantages when trying to model Sless mixins. A Sless mixin is essentially a sequence of declarations, where a selector `s1` is used as a handle, i.e. a mixin looks identical to a style rule. The mixin's declarations can be inserted into the sheet at any point by using function call-style syntax, i.e. `s1()`. Mixins can also be parameterized, i.e. provided with arguments just like real functions, and are then -unsurprisingly- called parametric mixins.

The selector in a mixin is often simply used as a hack for a function name and the parameters simply correspond to function parameters; hence the approach in this section to simply use Scala functions to model (parametric) mixins, instead of shoehorning them into our Sless AST syntax. Take a look at the `MixinDSL` file: it adds a single method `mixin` through implicit class syntax, that will allow us to flexibly create mixins through the inherent power of Scala's functions and values. Let's see an example of a regular mixin in Sless to get you started:

```
val asgn = prop("property-1") := value("value-1")
val mixin = List(asgn,asgn)
val propertiesPre = List(prop("property-pre") := value("value-pre"))
val propertiesPost = List()
val rule = All.mixin(propertiesPre, mixin, propertiesPost)
```

Here the mixin `mixin` gets mixed into the rule `rule`, after the list of properties `propertiesPre`. Parametric mixins work very similarly, but in that case `mixin` is a Scala function. First implement

the necessary mixin building blocks in `MixinImplementation`, use them to write your own additional tests in `MixinTest` and impress us with your mixing skills!

Nested style rules

DSLs: `NestedSelectorDSL`

Tests: `LessNestingImplementation`, `LessNestingTest`

In this section, we will extend our basic Sless ASTs in order to allow style rules to be nested inside the block of declarations of an outer style rule. The selectors of inner style rules will be able to reference their parents' selectors by using a new `Selector` constant `Parent`. To allow `Rules` to appear inside style rule bodies alongside `Declarations`, the `NestedSelectorDSL` provides a new sum-type `RuleOrDeclaration` that should accommodate both rules and declarations. The internal method `bindWithNesting` now allows creating a nested `Rule` given a `Selector` and a sequence of `RuleOrDeclarations`. The method `nest` provides implicit class notation.

For example, the following style rule is now legal:

```
(All ## "header").nest(
        (Parent |+ Parent)(
          prop("width") := value("300px")
        )
```

and gets flattened into:

```
*#header + *#header {
    width: 300px;
}
```

It is important to understand how the resulting selector is calculated for each style rule during flattening. When flattening selectors $s_1,\ldots,s_n$ the selector flattening algorithm works as follows, using an accumulative selector S:

1. Set $S=s_1$.
2. Repeat for each s_i in $s_2,\ldots,s_n$:
    1. If s_i does not contain `Parent`, add a descendant combinator, i.e. $S = S$ |- s_i
    2. If s_i contains `Parent`, substitute S for `Parent` in s_i, i.e. $S = s_i$[S/`Parent`]

To understand the algorithm better, have a look at (especially) the third test case we wrote. In case this explanation is unclear or still ambiguous to you, take a look at the exact same functionality in Less here, and consult the tests.

Additionally, keep in mind that even though the Sless ASTs now support nested style rules, the CSS code that is output by the `compile` and `pretty` methods in `Compilable` still does not, i.e. some `flatten` operation will now need to happen as part of (pretty) printing. For easy

compatibility with many other extensions, we advise you to perform this `flatten` operation early on during compilation.

Better Values

DSLs: `BetterValuesDSL`

In this extension you have to design your own mini-DSL. CSS declarations consist of different types of values bound to properties. However, in ValueDSL the only method available is a simple conversion from `String` to `Value`. Write your own mini-DSL to construct other types of values. This table has the full (original, and old) CSS property list, each property has its own set of accepted values. It is your task to implement the full `margin` group of values: https://www.w3.org/TR/CSS21/box.html#propdef-margin. A margin can take 1 to 4 margin widths as a value, or the `inherit` keyword. A `margin-width` in itself is either a length, percentage or the `auto` keyword. Make sure you support convenient Squants-like syntax for all values! E.g., we want to be able to write the following code:

```
prop("margin") := Margin(1.em, 2.px)
```

Make sure you define easy syntax for all different length units and for percentages. Create a test `BetterValuesTest` to show off and describe your features.

Do not forget to fill in `FILLME.md` to explain your implementation.

Extend

DSLs: `ExtendDSL`

Tests: `ExtendImplementation`, `ExtendTest`

In this extension, we add the option to extend selectors in our Sless sheets, similarly to what Less does. Concretely, we add a method `extend` that is applied to a `Selector s1`, and takes a `Selector s2` - that gets extended - as argument. The syntax in this case is `s1.extend(s2)`. Every other *exact* appearance of `s2` anywhere else in the sheet should now be replaced by a selector list `s2, s1`. If the same selector is extended in multiple locations, the extensions are added in order of appearance. The selector `s1` itself can also be a list selector, in which case you can consider `extend` to be applied to each element of the list. The extend-syntax itself just disappears during compilation. To spice things up, we dó allow (contrary to Less):

- extending *parts* of selectors, i.e. the `extend` call does not have to be the outermost selector
- nested `extend` calls to appear in `s1` (but *not* in `s2`, this is a regular selector): any outer `extend` call extends its selector after erasing any more inward `extend` calls in `s1`

Consult `ExtendTest` for more details on how to correctly implement `ExtendDSL`.

Go back and improve an existing feature

The title says it all: go back and do one or more of the following things:

- Implement specific pseudo-classes: at the moment, we only allow pseudo-classes that consist of a single `String` name. Some classes, e.g. nth-child, require arguments and are hence not yet correctly supported by our `compile` method. Pick a couple pseudo-classes that have arguments, implement them in your ASTs and design a DSL that allows instantiating them.
- Choose and implement more different, interesting lint checks (you can add your own methods to the DSL).
- Write an extensive and interesting testing example in your own test file, that showcases all the implementations you have implemented, and their interaction.

Do not forget to fill in `FILLME.md` to explain your implementation, and to always provide sufficient illustrating examples.

Come up with your own extension!

There is a whole range of Less-, Sass- and other extensions we have not yet covered in this assignment. Impress us by choosing and implementing your own extension, or be creative and make one up yourself.

Do not forget to fill in `FILLME.md` to document your implementation and any extra files you created.