Sass (Syntactically Awesome StyleSheets)

Sass is an extension of CSS that adds power and elegance to the basic language. It allows you to use variables, nested rules, mixins, inline imports, and more, all with a fully CSS-compatible syntax. Sass helps keep large stylesheets well-organized, and get small stylesheets up and running quickly, particularly with the help of the Compass style library.

Table of Contents (left)

Table of Contents (left)

1. Features
2. Syntax
3. Using Sass
    1. Rack/Rails/Merb Plugin
    2. Caching
    3. Options
    4. Syntax Selection
    5. Encodings
4. CSS Extensions
    1. Nested Rules
    2. Referencing Parent Selectors: &
    3. Nested Properties
    4. Placeholder Selectors: %foo
5. Comments: /* */ and //
6. SassScript
    1. Interactive Shell
    2. Variables: $
    3. Data Types
        1. Strings
        2. Lists
        3. Maps
        4. Colors
    4. Operations
        1. Number Operations
            1. Division and /
        2. Color Operations
        3. String Operations
        4. Boolean Operations
        5. List Operations
    5. Parentheses
    6. Functions
        1. Keyword Arguments
    7. Interpolation: #{}
    8. & in SassScript
    9. Variable Defaults: !default
7. @-Rules and Directives
    1. @import
        1. Partials
        2. Nested @import
    2. @media
    3. @extend
        1. How it Works
        2. Extending Complex Selectors
        3. Multiple Extends
        4. Chaining Extends
        5. Selector Sequences
            1. Merging Selector Sequences
        6. @extend-Only Selectors
        7. The !optional Flag
        8. @extend in Directives

Features

- Fully CSS3-compatible
- Language extensions such as variables, nesting, and mixins
- Many useful functions for manipulating colors and other values
- Advanced features like control directives for libraries
- Well-formatted, customizable output

Syntax

There are two syntaxes available for Sass. The first, known as SCSS (Sassy CSS) and used throughout this reference, is an extension of the syntax of CSS3. This means that every valid CSS3 stylesheet is a valid SCSS file with the same meaning. In addition, SCSS understands most CSS hacks and vendor-specific syntax, such as IE's old `filter` syntax. This syntax is enhanced with the Sass features described below. Files using this syntax have the `.scss` extension.

The second and older syntax, known as the indented syntax (or sometimes just "Sass"), provides a more concise way of writing CSS. It uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties. Some people find this to be easier to read and quicker to write than SCSS. The indented syntax has all the same features, although some of them have slightly different syntax; this is described in the indented syntax reference. Files using this syntax have the `.sass` extension.

Either syntax can import files written in the other. Files can be automatically converted from one syntax to the other using the `sass-convert` command line tool:

```
# Convert Sass to SCSS
$ sass-convert style.sass style.scss

# Convert SCSS to Sass
$ sass-convert style.scss style.sass
```

Note that this command does *not* generate CSS files. For that, use the `sass` command described elsewhere.

Using Sass

Sass 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. The first step for all of these is to install the Sass gem:

```
gem install sass
```

If you're using Windows, you may need to install Ruby first.

To run Sass from the command line, just use

```
sass input.scss output.css
```

You can also tell Sass to watch the file and update the CSS every time the Sass file changes:

```
sass --watch input.scss:output.css
```

If you have a directory with many Sass files, you can also tell Sass to watch the entire directory:

```
sass --watch app/sass:public/stylesheets
```

Use `sass --help` for full documentation.

Using Sass in Ruby code is very simple. After installing the Sass gem, you can use it by running `require "sass"` and using Sass::Engine like so:

```
engine = Sass::Engine.new("#main {background-color:
engine.render #=> "#main { background-color: #0000ff
```

Rack/Rails/Merb Plugin

To enable Sass in Rails versions before Rails 3, add the following line to `environment.rb`:

```
config.gem "sass"
```

For Rails 3, instead add the following line to the Gemfile:

```
gem "sass"
```

To enable Sass in Merb, add the following line to `config/dependencies.rb`:

4. @at-root
   1. @at-root (without: ...) and @at-root (with: ...)
5. @debug
6. @warn
7. @error

8. **Control Directives & Expressions**
   1. if()
   2. @if
   3. @for
   4. @each
      1. Multiple Assignment
   5. @while
9. **Mixin Directives**
   1. Defining a Mixin: @mixin
   2. Including a Mixin: @include
   3. Arguments
      1. Keyword Arguments
      2. Variable Arguments
   4. Passing Content Blocks to a Mixin
      1. Variable Scope and Content Blocks
10. **Function Directives**
11. **Output Style**
    1. :nested
    2. :expanded
    3. :compact
    4. :compressed
12. **Extending Sass**
    1. Defining Custom Sass Functions
    2. Cache Stores
    3. Custom Importers

```
dependency "merb-haml"
```

To enable Sass in a Rack application, add

```
require 'sass/plugin/rack'
use Sass::Plugin::Rack
```

to `config.ru`.

Sass stylesheets don't work the same as views. They don't contain dynamic content, so the CSS only needs to be generated when the Sass file has been updated. By default, `.sass` and `.scss` files are placed in public/stylesheets/sass (this can be customized with the `:template_location` option). Then, whenever necessary, they're compiled into corresponding CSS files in public/stylesheets. For instance, public/stylesheets/sass/main.scss would be compiled to public/stylesheets/main.css.

Caching

By default, Sass caches compiled templates and partials. This dramatically speeds up re-compilation of large collections of Sass files, and works best if the Sass templates are split up into separate files that are all `@import`ed into one large file.

Without a framework, Sass puts the cached templates in the `.sass-cache` directory. In Rails and Merb, they go in `tmp/sass-cache`. The directory can be customized with the `:cache_location` option. If you don't want Sass to use caching at all, set the `:cache` option to `false`.

Options

Options can be set by setting the Sass::Plugin#options hash in `environment.rb` in Rails or `config.ru` in Rack…

```
Sass::Plugin.options[:style] = :compact
```

…or by setting the `Merb::Plugin.config[:sass]` hash in `init.rb` in Merb…

```
Merb::Plugin.config[:sass][:style] = :compact
```

…or by passing an options hash to Sass::Engine#initialize. All relevant options are also available via flags to the `sass` and `scss` command-line executables. Available options are:

`:style`

Sets the style of the CSS output. See Output Style.

`:syntax`

The syntax of the input file, `:sass` for the indented syntax and `:scss` for the CSS-extension syntax. This is only useful when you're constructing Sass::Engine instances yourself; it's automatically set properly when using Sass::Plugin. Defaults to `:sass`.

`:property_syntax`

Forces indented-syntax documents to use one syntax for properties. If the correct syntax isn't used, an error is thrown. `:new` forces the use of a colon after the property name. For example: `color: #0f3` or `width: $main_width`. `:old` forces the use of a colon before the property name. For example: `:color #0f3` or `:width $main_width`. By default, either syntax is valid. This has no effect on SCSS documents.

`:cache`

Whether parsed Sass files should be cached, allowing greater speed. Defaults to true.

`:read_cache`

If this is set and `:cache` is not, only read the Sass cache if it exists, don't write to it if it doesn't.

`:cache_store`

If this is set to an instance of a subclass of Sass::CacheStores::Base, that cache store will be used to store and retrieve cached compilation results. Defaults to a Sass::CacheStores::Filesystem that is initialized using the `:cache_location` option.

`:never_update`

Whether the CSS files should never be updated, even if the template file changes. Setting this to true may give small performance gains. It always defaults to false. Only has meaning within Rack, Ruby on Rails, or Merb.

`:always_update`

Whether the CSS files should be updated every time a controller is accessed, as opposed to only when the template has been modified. Defaults to false. Only has meaning within Rack, Ruby on Rails, or Merb.

`:always_check`

Whether a Sass template should be checked for updates every time a controller is accessed, as opposed to only when the server starts. If a Sass template has been updated, it will be recompiled and will overwrite the corresponding CSS file. Defaults to false in production mode, true otherwise. Only has meaning within Rack, Ruby on Rails, or Merb.

`:poll`

When true, always use the polling backend for Sass::Plugin::Compiler#watch rather than the native filesystem backend.

`:full_exception`

Whether an error in the Sass code should cause Sass to provide a detailed description within the generated CSS file. If set to true, the error will be displayed along with a line number and source snippet both as a comment in the CSS file and at the top of the page (in supported browsers). Otherwise, an exception will be raised in the Ruby code. Defaults to false in production mode, true otherwise.

`:template_location`

A path to the root sass template directory for your application. If a hash, `:css_location` is ignored and this option designates a mapping between input and output directories. May also be given a list of 2-element lists, instead of a hash. Defaults to `css_location + "/sass"`. Only has meaning within Rack, Ruby on Rails, or Merb. Note that if multiple template locations are specified, all of them are placed in the import path, allowing you to import between them. **Note that due to the many possible formats it can take, this option should only be set directly, not accessed or modified. Use the Sass::Plugin#template_location_array, Sass::Plugin#add_template_location, and Sass::Plugin#remove_template_location methods instead**.

`:css_location`

The path where CSS output should be written to. This option is ignored when `:template_location` is a Hash. Defaults to `"./public/stylesheets"`. Only has meaning within Rack, Ruby on Rails, or Merb.

`:cache_location`

The path where the cached `sassc` files should be written to. Defaults to `"./tmp/sass-cache"` in Rails and Merb, or `"./.sass-cache"` otherwise. If the

`:cache_store` option is set, this is ignored.

`:unix_newlines`

If true, use Unix-style newlines when writing files. Only has meaning on Windows, and only when Sass is writing the files (in Rack, Rails, or Merb, when using Sass::Plugin directly, or when using the command-line executable).

`:filename`

The filename of the file being rendered. This is used solely for reporting errors, and is automatically set when using Rack, Rails, or Merb.

`:line`

The number of the first line of the Sass template. Used for reporting line numbers for errors. This is useful to set if the Sass template is embedded in a Ruby file.

`:load_paths`

An array of filesystem paths or importers which should be searched for Sass templates imported with the `@import` directive. These may be strings, `Pathname` objects, or subclasses of Sass::Importers::Base. This defaults to the working directory and, in Rack, Rails, or Merb, whatever `:template_location` is. The load path is also informed by Sass.load_paths and the `SASS_PATH` environment variable.

`:filesystem_importer`

A Sass::Importers::Base subclass used to handle plain string load paths. This should import files from the filesystem. It should be a Class object inheriting from Sass::Importers::Base with a constructor that takes a single string argument (the load path). Defaults to Sass::Importers::Filesystem.

`:sourcemap`

Controls how sourcemaps are generated. These sourcemaps tell the browser how to find the Sass styles that caused each CSS style to be generated. This has three valid values: **`:auto`** uses relative URIs where possible, assuming that that the source stylesheets will be made available on whatever server you're using, and that their relative location will be the same as it is on the local filesystem. If a relative URI is unavailable, a "file:" URI is used instead. **`:file`** always uses "file:" URIs, which will work locally but can't be deployed to a remote server. **`:inline`** includes the full source text in the sourcemap, which is maximally portable but can create very large sourcemap files. Finally, **`:none`** causes no sourcemaps to be generated at all.

`:line_numbers`

When set to true, causes the line number and file where a selector is defined to be emitted into the compiled CSS as a comment. Useful for debugging, especially when using imports and mixins. This option may also be called `:line_comments`. Automatically disabled when using the `:compressed` output style or the `:debug_info`/`:trace_selectors` options.

`:trace_selectors`

When set to true, emit a full trace of imports and mixins before each selector. This can be helpful for in-browser debugging of stylesheet imports and mixin includes. This option supersedes the `:line_comments` option and is superseded by the `:debug_info` option. Automatically disabled when using the `:compressed` output style.

`:debug_info`

When set to true, causes the line number and file where a selector is defined to be emitted into the compiled CSS in a format that can be understood by the browser. Useful in conjunction with the FireSass Firebug extension for displaying the Sass filename and line number. Automatically disabled when using the `:compressed` output style.

`:custom`

An option that's available for individual applications to set to make data available to custom Sass functions.

`:quiet`

When set to true, causes warnings to be disabled.

Syntax Selection

The Sass command-line tool will use the file extension to determine which syntax you are using, but there's not always a filename. The `sass` command-line program defaults to the indented syntax but you can pass the `--scss` option to it if the input should be interpreted as SCSS syntax. Alternatively, you can use the `scss` command-line program which is exactly like the `sass` program but it defaults to assuming the syntax is SCSS.

Encodings

When running on Ruby 1.9 and later, Sass is aware of the character encoding of documents. Sass follows the CSS spec to determine the encoding of a stylesheet, and falls back to the Ruby string encoding. This means that it first checks the Unicode byte order mark, then the `@charset` declaration, then the Ruby string encoding. If none of these are set, it will assume the document is in UTF-8.

To explicitly specify the encoding of your stylesheet, use a `@charset` declaration just like in CSS. Add `@charset "encoding-name";` at the beginning of the stylesheet (before any whitespace or comments) and Sass will interpret it as the given encoding. Note that whatever encoding you use, it must be convertible to Unicode.

Sass will always encode its output as UTF-8. It will include a `@charset` declaration if and only if the output file contains non-ASCII characters. In compressed mode, a UTF-8 byte order mark is used in place of a `@charset` declaration.

CSS Extensions

Nested Rules

Sass allows CSS rules to be nested within one another. The inner rule then only applies within the outer rule's selector. For example:

```
#main p {
  color: #00ff00;
  width: 97%;

  .redbox {
    background-color: #ff0000;
    color: #000000;
  }
}
```

is compiled to:

```
#main p {
  color: #00ff00;
  width: 97%; }
  #main p .redbox {
    background-color: #ff0000;
    color: #000000; }
```

This helps avoid repetition of parent selectors, and makes complex CSS layouts with lots of nested selectors much simpler. For example:

```
#main {
  width: 97%;

  p, div {
    font-size: 2em;
    a { font-weight: bold; }
  }

  pre { font-size: 3em; }
}
```

is compiled to:

```
#main {
  width: 97%; }
  #main p, #main div {
    font-size: 2em; }
    #main p a, #main div a {
      font-weight: bold; }
  #main pre {
    font-size: 3em; }
```

Referencing Parent Selectors: `&`

Sometimes it's useful to use a nested rule's parent selector in other ways than the default. For instance, you might want to have special styles for when that selector is hovered over or for when the body element has a certain class. In these cases, you can explicitly specify where the parent selector should be inserted using the `&` character. For example:

```
a {
  font-weight: bold;
  text-decoration: none;
  &:hover { text-decoration: underline; }
  body.firefox & { font-weight: normal; }
}
```

is compiled to:

```
a {
  font-weight: bold;
  text-decoration: none; }
  a:hover {
    text-decoration: underline; }
  body.firefox a {
    font-weight: normal; }
```

`&` will be replaced with the parent selector as it appears in the CSS. This means that if you have a deeply nested rule, the parent selector will be fully resolved before the `&` is replaced. For example:

```
#main {
  color: black;
  a {
    font-weight: bold;
    &:hover { color: red; }
  }
}
```

is compiled to:

```
#main {
  color: black; }
  #main a {
    font-weight: bold; }
    #main a:hover {
      color: red; }
```

`&` must appear at the beginning of a compound selector, but it can be followed by a suffix that will be added to the parent selector. For example:

```
#main {
  color: black;
  &-sidebar { border: 1px solid; }
}
```

is compiled to:

```
#main {
  color: black; }
  #main-sidebar {
    border: 1px solid; }
```

If the parent selector can't have a suffix applied, Sass will throw an error.

Nested Properties

CSS has quite a few properties that are in “namespaces;” for instance, `font-family`, `font-size`, and `font-weight` are all in the `font` namespace. In CSS, if you want to set a bunch of properties in the same namespace, you have to type it out each time. Sass provides a shortcut for this: just write the namespace once, then nest each of the sub-properties within it. For example:

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

is compiled to:

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

The property namespace itself can also have a value. For example:

```
.funky {
  font: 20px/24px fantasy {
    weight: bold;
  }
}
```

is compiled to:

```
.funky {
  font: 20px/24px fantasy;
    font-weight: bold;
}
```

Placeholder Selectors: `%foo`

Sass supports a special type of selector called a "placeholder selector". These look like class and id selectors, except the `#` or `.` is replaced by `%`. They're meant to be used with the `@extend` directive; for more information see `@extend`-Only Selectors.

On their own, without any use of `@extend`, rulesets that use placeholder selectors will not be rendered to CSS.

Comments: `/* */` and `//`

Sass supports standard multiline CSS comments with `/* */`, as well as single-line comments with `//`. The multiline comments are preserved in the CSS output where possible, while the single-line comments are removed. For example:

```
/* This comment is
 * several lines long.
 * since it uses the CSS comment syntax,
 * it will appear in the CSS output. */
body { color: black; }

// These comments are only one line long each.
// They won't appear in the CSS output,
// since they use the single-line comment syntax.
a { color: green; }
```

is compiled to:

```
/* This comment is
 * several lines long.
 * since it uses the CSS comment syntax,
 * it will appear in the CSS output. */
body {
  color: black; }

a {
  color: green; }
```

When the first letter of a multiline comment is `!`, the comment will always rendered into css output even in compressed output modes. This is useful for adding Copyright notices to your generated CSS.

Since multiline comments become part of the resulting CSS, interpolation within them is resolved. For example:

```
$version: "1.2.3";
/* This CSS is generated by My Snazzy Framework version #{$version}. */
```

is compiled to:

```
$version: "1.2.3";
/* This CSS is generated by My Snazzy Framework version 1.2.3. */
```

SassScript

In addition to the plain CSS property syntax, Sass supports a small set of extensions called SassScript. SassScript allows properties to use variables, arithmetic, and extra functions. SassScript can be used in any property value.

SassScript can also be used to generate selectors and property names, which is useful when writing mixins. This is done via interpolation.

Interactive Shell

You can easily experiment with SassScript using the interactive shell. To launch the shell run the sass command-line with the `-i` option. At the prompt, enter any legal SassScript expression to have it evaluated and the result printed out for you:

```
$ sass -i
>> "Hello, Sassy World!"
"Hello, Sassy World!"
>> 1px + 1px + 1px
3px
>> #777 + #777
#eeeeee
>> #777 + #888
white
```

Variables: `$`

The most straightforward way to use SassScript is to use variables. Variables begin with dollar signs, and are set like CSS properties:

```
$width: 5em;
```

You can then refer to them in properties:

```
#main {
  width: $width;
}
```

Variables are only available within the level of nested selectors where they're defined. If they're defined outside of any nested selectors, they're available everywhere. They can also be defined with the `!global` flag, in which case they're also available everywhere. For example:

```
#main {
  $width: 5em !global;
  width: $width;
}

#sidebar {
  width: $width;
}
```

is compiled to:

```
#main {
  width: 5em;
}

#sidebar {
  width: 5em;
}
```

Data Types

SassScript supports seven 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)`)
- booleans (e.g. `true`, `false`)
- nulls (e.g. `null`)
- lists of values, separated by spaces or commas (e.g. `1.5em 1em 0 2em`, `Helvetica, Arial, sans-serif`)
- maps from one value to another (e.g. `(key1: value1, key2: value2)`)

SassScript also supports all other types of CSS property value, such as Unicode ranges and `!important` declarations. However, it has no special handling for these types. They're treated just like unquoted strings.

Strings

CSS specifies two kinds of strings: those with quotes, such as `"Lucida Grande"` or `'http://sass-lang.com'`, and those without quotes, such as `sans-serif` or `bold`. SassScript recognizes both kinds, and in general if one kind of string is used in the Sass document, that kind of string will be used in the resulting CSS.

There is one exception to this, though: when using `#{}` interpolation, quoted strings are unquoted. This makes it easier to use e.g. selector names in mixins. For example:

```
@mixin firefox-message($selector) {
  body.firefox #{$selector}:before {
    content: "Hi, Firefox users!";
  }
}

@include firefox-message(".header");
```

is compiled to:

```
body.firefox .header:before {
  content: "Hi, Firefox users!"; }
```

Lists

Lists are how Sass represents the values of CSS declarations like `margin: 10px 15px 0 0` or `font-face: Helvetica, Arial, sans-serif`. Lists are just a series of other values, separated by either spaces or commas. In fact, individual values count as lists, too: they're just lists with one item.

On their own, lists don't do much, but the SassScript list functions make them useful. The `nth` function can access items in a list, the `join` function can join multiple lists together, and the `append` function can add items to lists. The `@each` directive can also add styles for each item in a list.

In addition to containing simple values, lists can contain other lists. For example, `1px 2px, 5px 6px` is a two-item list containing the list `1px 2px` and the list `5px 6px`. If the inner lists have the same separator as the outer list, you'll need to use parentheses to make it clear where the inner lists start and stop. For example, `(1px 2px) (5px 6px)` is also a two-item list containing the list `1px 2px` and the list `5px 6px`. The difference is that the outer list is space-separated, where before it was comma-separated.

When lists are turned into plain CSS, Sass doesn't add any parentheses, since CSS doesn't understand them. That means that `(1px 2px) (5px 6px)` and `1px 2px 5px 6px` will look the same when they become CSS. However, they aren't the same when they're Sass: the first is a list containing two lists, while the second is a list containing four numbers.

Lists can also have no items in them at all. These lists are represented as `()` (which is also an empty map). They can't be output directly to CSS; if you try to do e.g. `font-family: ()`, Sass will raise an error. If a list contains empty lists or null values, as in `1px 2px () 3px` or `1px 2px null 3px`, the empty lists and null values will be removed before the containing list is turned into CSS.

Comma-separated lists may have a trailing comma. This is especially useful because it allows you to represent a single-element list. For example, `(1,)` is a list containing `1` and `(1 2 3,)` is a comma-separated list containing a space-separated list containing `1`, `2`, and `3`.

Maps

Maps represent an association between keys and values, where keys are used to look up values. They make it easy to collect values into named groups and access those groups dynamically. They have no direct parallel in CSS, although they're syntactically similar to media query expressions:

```
$map: (key1: value1, key2: value2, key3: value3);
```

Unlike lists, maps must always be surrounded by parentheses and must always be comma-separated. Both the keys and values in maps can be any SassScript object. A map may only have one value associated with a given key (although that value may be a list). A given value may be associated with many keys, though.

Like lists, maps are mostly manipulated using SassScript functions. The `map-get` function looks up values in a map and the `map-merge` function adds values to a map. The `@each` directive can be used to add styles for each key/value pair in a map. The order of pairs in a map is always the same as when the map was created.

Maps can also be used anywhere lists can. When used by a list function, a map is treated as a list of pairs. For example, `(key1: value1, key2: value2)` would be treated as the nested list `key1 value1, key2 value2` by list functions. Lists cannot be treated as maps, though, with the exception of the empty list. `()` represents both a map with no key/value pairs and a list with no elements.

Note that map keys can be any Sass data type (even another map) and the syntax for declaring a map allows arbitrary SassScript expressions that will be evaluated to determine the key.

Maps cannot be converted to plain CSS. Using one as the value of a variable or an argument to a CSS function will cause an error. Use the `inspect($value)` function to produce an output string useful for debugging maps.

Colors

Any CSS color expression returns a SassScript Color value. This includes a large number of named colors which are indistinguishable from unquoted strings.

In compressed output mode, Sass will output the smallest CSS representation of a color. For example, `#FF0000` will output as `red` in compressed mode, but `blanchedalmond` will output as `#FFEBCD`.

A common issue users encounter with named colors is that since Sass prefers the same output format as was typed in other output modes, a color interpolated into a selector becomes invalid syntax when compressed. To avoid this, always quote named colors if they are meant to be used in the construction of a selector.

Operations

All types support equality operations (`==` and `!=`). In addition, each type has its own operations that it has special support for.

Number Operations

SassScript supports the standard arithmetic operations on numbers (addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`). Sass math functions preserve units during arithmetic operations. This means that, just like in real life, you cannot work on numbers with incompatible units (such as adding a number with `px` and `em`) and two numbers with the same unit that are multiplied together will produce square units (`10px * 10px == 100px * px`). **Be Aware** that `px * px` is an invalid CSS unit and you will get an error from Sass for attempting to use invalid units in CSS.

Relational operators (`<`, `>`, `<=`, `>=`) are also supported for numbers, and equality operators (`==`, `!=`) are supported for all types.

Division and `/`

CSS allows `/` to appear in property values as a way of separating numbers. Since SassScript is an extension of the CSS property syntax, it must support this, while also allowing `/` to be used for division. This means that by default, if two numbers are separated by `/` in SassScript, then they will appear that way in the resulting CSS.

However, there are three situations where the `/` will be interpreted as division. These cover the vast majority of cases where division is actually used. They are:

1. If the value, or any part of it, is stored in a variable or returned by a function.
2. If the value is surrounded by parentheses.
3. If the value is used as part of another arithmetic expression.

For example:

```
p {
  font: 10px/8px;             // Plain CSS, no division
  $width: 1000px;
  width: $width/2;            // Uses a variable, does division
  width: round(1.5)/2;        // Uses a function, does division
  height: (500px/2);          // Uses parentheses, does division
  margin-left: 5px + 8px/2px; // Uses +, does division
}
```

is compiled to:

```
p {
  font: 10px/8px;
  width: 500px;
  height: 250px;
  margin-left: 9px; }
```

If you want to use variables along with a plain CSS `/`, you can use `#{}` to insert them. For example:

```
p {
  $font-size: 12px;
  $line-height: 30px;
  font: #{$font-size}/#{$line-height};
}
```

is compiled to:

```
p {
  font: 12px/30px; }
```

Color Operations

All arithmetic operations are supported for color values, where they work piecewise. This means that the operation is performed on the red, green, and blue components in turn. For example:

```
p {
  color: #010203 + #040506;
}
```

computes `01 + 04 = 05`, `02 + 05 = 07`, and `03 + 06 = 09`, and is compiled to:

```
p {
  color: #050709; }
```

Often it's more useful to use color functions than to try to use color arithmetic to achieve the same effect.

Arithmetic operations also work between numbers and colors, also piecewise. For example:

```
p {
  color: #010203 * 2;
}
```

computes `01 * 2 = 02`, `02 * 2 = 04`, and `03 * 2 = 06`, and is compiled to:

```
p {
  color: #020406; }
```

Note that colors with an alpha channel (those created with the rgba or hsla functions) must have the same alpha value in order for color arithmetic to be done with them. The arithmetic doesn't affect the alpha value. For example:

```
p {
  color: rgba(255, 0, 0, 0.75) + rgba(0, 255, 0, 0.75);
}
```

is compiled to:

```
p {
  color: rgba(255, 255, 0, 0.75); }
```

The alpha channel of a color can be adjusted using the opacify and transparentize functions. For example:

```
$translucent-red: rgba(255, 0, 0, 0.5);
p {
  color: opacify($translucent-red, 0.3);
  background-color: transparentize($translucent-red, 0.25);
}
```

is compiled to:

```
p {
  color: rgba(255, 0, 0, 0.8);
  background-color: rgba(255, 0, 0, 0.25); }
```

IE filters require all colors include the alpha layer, and be in the strict format of #AABBCCDD. You can more easily convert the color using the ie_hex_str function. For example:

```
$translucent-red: rgba(255, 0, 0, 0.5);
$green: #00ff00;
div {
  filter: progid:DXImageTransform.Microsoft.gradient(enabled='false', startColorstr='#{ie-hex-str($green)}', endColorstr='#{ie-hex-str($
}
```

is compiled to:

```
div {
  filter: progid:DXImageTransform.Microsoft.gradient(enabled='false', startColorstr=#FF00FF00, endColorstr=#80FF0000);
}
```

String Operations

The + operation can be used to concatenate strings:

```
p {
  cursor: e + -resize;
}
```

is compiled to:

```
p {
  cursor: e-resize; }
```

Note that if a quoted string is added to an unquoted string (that is, the quoted string is to the left of the +), the result is a quoted string. Likewise, if an unquoted string is added to a quoted string (the unquoted string is to the left of the +), the result is an unquoted string. For example:

```
p:before {
  content: "Foo " + Bar;
  font-family: sans- + "serif";
}
```

is compiled to:

```
p:before {
  content: "Foo Bar";
  font-family: sans-serif; }
```

By default, if two values are placed next to one another, they are concatenated with a space:

```
p {
  margin: 3px + 4px auto;
}
```

is compiled to:

```
p {
  margin: 7px auto; }
```

Within a string of text, #{} style interpolation can be used to place dynamic values within the string:

```
p:before {
  content: "I ate #{5 + 10} pies!";
}
```

is compiled to:

```
p:before {
  content: "I ate 15 pies!"; }
```

Null values are treated as empty strings for string interpolation:

```
$value: null;
p:before {
  content: "I ate #{$value} pies!";
}
```

is compiled to:

```
p:before {
  content: "I ate  pies!"; }
```

Boolean Operations

SassScript supports `and`, `or`, and `not` operators for boolean values.

List Operations

Lists don't support any special operations. Instead, they're manipulated using the list functions.

Parentheses

Parentheses can be used to affect the order of operations:

```
p {
  width: 1em + (2em * 3);
}
```

is compiled to:

```
p {
  width: 7em; }
```

Functions

SassScript defines some useful functions that are called using the normal CSS function syntax:

```
p {
  color: hsl(0, 100%, 50%);
}
```

is compiled to:

```
p {
  color: #ff0000; }
```

Keyword Arguments

Sass functions can also be called using explicit keyword arguments. The above example can also be written as:

```
p {
  color: hsl($hue: 0, $saturation: 100%, $lightness: 50%);
}
```

While this is less concise, it can make the stylesheet easier to read. It also allows functions to present more flexible interfaces, providing many arguments without becoming difficult to call.

Named arguments can be passed in any order, and arguments with default values can be omitted. Since the named arguments are variable names, underscores and dashes can be used interchangeably.

See Sass::Script::Functions for a full listing of Sass functions and their argument names, as well as instructions on defining your own in Ruby.

Interpolation: `#{}`

You can also use SassScript variables in selectors and property names using `#{}` interpolation syntax:

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

is compiled to:

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

It's also possible to use `#{}` to put SassScript into property values. In most cases this isn't any better than using a variable, but using `#{}` does mean that any operations near it will be treated as plain CSS. For example:

```
p {
  $font-size: 12px;
  $line-height: 30px;
  font: #{$font-size}/#{$line-height};
}
```

is compiled to:

```
p {
  font: 12px/30px; }
```

`&` in SassScript

Just like when it's used in selectors, `&` in SassScript refers to the current parent selector. It's a comma-separated list of space-separated lists. For example:

```
.foo.bar .baz.bang, .bip.qux {
  $selector: &;
}
```

The value of `$selector` is now `((".foo.bar" ".baz.bang"), ".bip.qux")`. The compound selectors are quoted here to indicate that they're strings, but in reality they would be unquoted. Even if the parent selector doesn't contain a comma or a space, `&` will always have two levels of nesting, so it can be accessed consistently.

If there is no parent selector, the value of `&` will be null. This means you can use it in a mixin to detect whether a parent selector exists:

```
@mixin does-parent-exist {
  @if & {
    &:hover {
      color: red;
    }
  } @else {
    a {
      color: red;
    }
  }
}
```

Variable Defaults: `!default`

You can assign to variables if they aren't already assigned by adding the `!default` flag to the end of the value. This means that if the variable has already been assigned to, it won't be re-assigned, but if it doesn't have a value yet, it will be given one.

For example:

```
$content: "First content";
$content: "Second content?" !default;
$new_content: "First time reference" !default;

#main {
  content: $content;
  new-content: $new_content;
}
```

is compiled to:

```
#main {
  content: "First content";
  new-content: "First time reference"; }
```

Variables with `null` values are treated as unassigned by !default:

```
$content: null;
$content: "Non-null content" !default;

#main {
  content: $content;
}
```

is compiled to:

```
#main {
  content: "Non-null content"; }
```

@-Rules and Directives

Sass supports all CSS3 @-rules, as well as some additional Sass-specific ones known as "directives." These have various effects in Sass, detailed below. See also control directives and mixin directives.

@import

Sass extends the CSS @import rule to allow it to import SCSS and Sass files. All imported SCSS and Sass files will be merged together into a single CSS output file. In addition, any variables or mixins defined in imported files can be used in the main file.

Sass looks for other Sass files in the current directory, and the Sass file directory under Rack, Rails, or Merb. Additional search directories may be specified using the :load_paths option, or the --load-path option on the command line.

@import takes a filename to import. By default, it looks for a Sass file to import directly, but there are a few circumstances under which it will compile to a CSS @import rule:

- If the file's extension is .css.
- If the filename begins with http://.
- If the filename is a url().
- If 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 the .scss or .sass extension and import it.

For example,

```
@import "foo.scss";
```

or

```
@import "foo";
```

would both import the file foo.scss, whereas

```
@import "foo.css";
@import "foo" screen;
@import "http://foo.com/bar";
@import url(foo);
```

would all compile to

```
@import "foo.css";
@import "foo" screen;
@import "http://foo.com/bar";
@import url(foo);
```

It's also possible to import multiple files in one @import. For example:

```
@import "rounded-corners", "text-shadow";
```

would import both the rounded-corners and the text-shadow files.

Imports may contain #{} interpolation, but only with certain restrictions. It's not possible to dynamically import a Sass file based on a variable; interpolation is only for CSS imports. As such, it only works with url() imports. For example:

```
$family: unquote("Droid+Sans");
@import url("http://fonts.googleapis.com/css?family=#{$family}");
```

would compile to

```
@import url("http://fonts.googleapis.com/css?family=Droid+Sans");
```

Partials

If you have a SCSS or Sass file that you want to import but don't want to compile to a CSS file, you can add an underscore to the beginning of the filename. This will tell Sass not to compile it to a normal CSS file. You can then import these files without using the underscore.

For example, you might have `_colors.scss`. Then no `_colors.css` file would be created, and you can do

```
@import "colors";
```

and `_colors.scss` would be imported.

Note that you may not include a partial and a non-partial with the same name in the same directory. For example, `_colors.scss` may not exist alongside `colors.scss`.

Nested `@import`

Although most of the time it's most useful to just have `@import`s at the top level of the document, it is possible to include them within CSS rules and `@media` rules. Like a base-level `@import`, this includes the contents of the `@import`ed file. However, the imported rules will be nested in the same place as the original `@import`.

For example, if `example.scss` contains

```
.example {
  color: red;
}
```

then

```
#main {
  @import "example";
}
```

would compile to

```
#main .example {
  color: red;
}
```

Directives that are only allowed at the base level of a document, like `@mixin` or `@charset`, are not allowed in files that are `@import`ed in a nested context.

It's not possible to nest `@import` within mixins or control directives.

`@media`

`@media` directives in Sass behave just like they do in plain CSS, with one extra capability: they can be nested in CSS rules. If a `@media` directive appears within a CSS rule, it will be bubbled up to the top level of the stylesheet, putting all the selectors on the way inside the rule. This makes it easy to add media-specific styles without having to repeat selectors or break the flow of the stylesheet. For example:

```
.sidebar {
  width: 300px;
  @media screen and (orientation: landscape) {
    width: 500px;
  }
}
```

is compiled to:

```
.sidebar {
  width: 300px; }
  @media screen and (orientation: landscape) {
    .sidebar {
      width: 500px; } }
```

`@media` queries can also be nested within one another. The queries will then be combined using the `and` operator. For example:

```
@media screen {
  .sidebar {
    @media (orientation: landscape) {
      width: 500px;
    }
  }
}
```

is compiled to:

```
@media screen and (orientation: landscape) {
  .sidebar {
    width: 500px; } }
```

Finally, `@media` queries can contain SassScript expressions (including variables, functions, and operators) in place of the feature names and feature values. For example:

```
$media: screen;
$feature: -webkit-min-device-pixel-ratio;
$value: 1.5;

@media #{$media} and ($feature: $value) {
  .sidebar {
    width: 500px;
  }
}
```

is compiled to:

```
@media screen and (-webkit-min-device-pixel-ratio: 1.5) {
  .sidebar {
    width: 500px; } }
```

`@extend`

There are often cases when designing a page when one class should have all the styles of another class, as well as its own specific styles. The most common way of handling this is to use both the more general class and the more specific class in the HTML. For example, suppose we have a design for a normal error and also for a serious error. We might write our markup like so:

```
<div class="error seriousError">
  Oh no! You've been hacked!
</div>
```

And our styles like so:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}
.seriousError {
  border-width: 3px;
}
```

Unfortunately, this means that we have to always remember to use `.error` with `.seriousError`. This is a maintenance burden, leads to tricky bugs, and can bring non-semantic style concerns into the markup.

The `@extend` directive avoids these problems by telling Sass that one selector should inherit the styles of another selector. For example:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}
.seriousError {
  @extend .error;
  border-width: 3px;
}
```

is compiled to:

```
.error, .seriousError {
  border: 1px #f00;
  background-color: #fdd;
}

.seriousError {
  border-width: 3px;
}
```

This means that all styles defined for `.error` are also applied to `.seriousError`, in addition to the styles specific to `.seriousError`. In effect, every element with class `.seriousError` also has class `.error`.

Other rules that use `.error` will work for `.seriousError` as well. For example, if we have special styles for errors caused by hackers:

```
.error.intrusion {
  background-image: url("/image/hacked.png");
}
```

Then `<div class="seriousError intrusion">` will have the `hacked.png` background image as well.

How it Works

`@extend` works by inserting the extending selector (e.g. `.seriousError`) anywhere in the stylesheet that the extended selector (.e.g `.error`) appears. Thus the example above:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}
.error.intrusion {
  background-image: url("/image/hacked.png");
}
.seriousError {
  @extend .error;
  border-width: 3px;
}
```

is compiled to:

```
.error, .seriousError {
  border: 1px #f00;
  background-color: #fdd; }

.error.intrusion, .seriousError.intrusion {
  background-image: url("/image/hacked.png"); }

.seriousError {
  border-width: 3px; }
```

When merging selectors, `@extend` is smart enough to avoid unnecessary duplication, so something like `.seriousError.seriousError` gets translated to `.seriousError`. In addition, it won't produce selectors that can't match anything, like `#main#footer`.

Extending Complex Selectors

Class selectors aren't the only things that can be extended. It's possible to extend any selector involving only a single element, such as `.special.cool`, `a:hover`, or `a.user[href^="http://"]`. For example:

```
.hoverlink {
  @extend a:hover;
}
```

Just like with classes, this means that all styles defined for `a:hover` are also applied to `.hoverlink`. For example:

```
.hoverlink {
  @extend a:hover;
}
a:hover {
  text-decoration: underline;
}
```

is compiled to:

```
a:hover, .hoverlink {
  text-decoration: underline; }
```

Just like with `.error.intrusion` above, any rule that uses `a:hover` will also work for `.hoverlink`, even if they have other selectors as well. For example:

```
.hoverlink {
  @extend a:hover;
}
.comment a.user:hover {
  font-weight: bold;
}
```

is compiled to:

```
.comment a.user:hover, .comment .user.hoverlink {
  font-weight: bold; }
```

Multiple Extends

A single selector can extend more than one selector. This means that it inherits the styles of all the extended selectors. For example:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}
.attention {
  font-size: 3em;
  background-color: #ff0;
}
.seriousError {
  @extend .error;
  @extend .attention;
  border-width: 3px;
}
```

is compiled to:

```
.error, .seriousError {
  border: 1px #f00;
  background-color: #fdd; }

.attention, .seriousError {
  font-size: 3em;
  background-color: #ff0; }

.seriousError {
  border-width: 3px; }
```

In effect, every element with class `.seriousError` also has class `.error` *and* class `.attention`. Thus, the styles defined later in the document take precedence: `.seriousError` has background color `#ff0` rather than `#fdd`, since `.attention` is defined later than `.error`.

Multiple extends can also be written using a comma-separated list of selectors. For example, `@extend .error, .attention` is the same as `@extend .error; @extend .attention`.

Chaining Extends

It's possible for one selector to extend another selector that in turn extends a third. For example:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}
.seriousError {
  @extend .error;
  border-width: 3px;
}
.criticalError {
  @extend .seriousError;
  position: fixed;
  top: 10%;
  bottom: 10%;
  left: 10%;
  right: 10%;
}
```

Now everything with class `.seriousError` also has class `.error`, and everything with class `.criticalError` has class `.seriousError` *and* class `.error`. It's compiled to:

```
.error, .seriousError, .criticalError {
  border: 1px #f00;
  background-color: #fdd; }

.seriousError, .criticalError {
  border-width: 3px; }

.criticalError {
  position: fixed;
  top: 10%;
  bottom: 10%;
  left: 10%;
  right: 10%; }
```

Selector Sequences

Selector sequences, such as `.foo .bar` or `.foo + .bar`, currently can't be extended. However, it is possible for nested selectors themselves to use `@extend`. For example:

```
#fake-links .link {
  @extend a;
}

a {
  color: blue;
  &:hover {
    text-decoration: underline;
  }
}
```

is compiled to

```
a, #fake-links .link {
  color: blue; }
  a:hover, #fake-links .link:hover {
    text-decoration: underline; }
```

Merging Selector Sequences

Sometimes a selector sequence extends another selector that appears in another sequence. In this case, the two sequences need to be merged. For example:

```
#admin .tabbar a {
  font-weight: bold;
}
#demo .overview .fakelink {
  @extend a;
}
```

While it would technically be possible to generate all selectors that could possibly match either sequence, this would make the stylesheet far too large. The simple example above, for instance, would require ten selectors. Instead, Sass generates only selectors that are likely to be useful.

When the two sequences being merged have no selectors in common, then two new selectors are generated: one with the first sequence before the second, and one with the second sequence before the first. For example:

```
#admin .tabbar a {
  font-weight: bold;
}
#demo .overview .fakelink {
  @extend a;
}
```

is compiled to:

```
#admin .tabbar a,
#admin .tabbar #demo .overview .fakelink,
#demo .overview #admin .tabbar .fakelink {
  font-weight: bold; }
```

If the two sequences do share some selectors, then those selectors will be merged together and only the differences (if any still exist) will alternate. In this example, both sequences contain the id `#admin`, so the resulting selectors will merge those two ids:

```
#admin .tabbar a {
  font-weight: bold;
}
#admin .overview .fakelink {
  @extend a;
}
```

This is compiled to:

```
#admin .tabbar a,
#admin .tabbar .overview .fakelink,
#admin .overview .tabbar .fakelink {
  font-weight: bold; }
```

`@extend`-Only Selectors

Sometimes you'll write styles for a class that you only ever want to `@extend`, and never want to use directly in your HTML. This is especially true when writing a Sass library, where you may provide styles for users to `@extend` if they need and ignore if they don't.

If you use normal classes for this, you end up creating a lot of extra CSS when the stylesheets are generated, and run the risk of colliding with other classes that are being used in the HTML. That's why Sass supports "placeholder selectors" (for example, `%foo`).

Placeholder selectors look like class and id selectors, except the `#` or `.` is replaced by `%`. They can be used anywhere a class or id could, and on their own they prevent rulesets from being rendered to CSS. For example:

```
// This ruleset won't be rendered on its own.
#context a%extreme {
  color: blue;
  font-weight: bold;
  font-size: 2em;
}
```

However, placeholder selectors can be extended, just like classes and ids. The extended selectors will be generated, but the base placeholder selector will not. For example:

```
.notice {
  @extend %extreme;
}
```

Is compiled to:

```
#context a.notice {
  color: blue;
  font-weight: bold;
  font-size: 2em; }
```

The `!optional` Flag

Normally when you extend a selector, it's an error if that `@extend` doesn't work. For example, if you write `a.important {@extend .notice}`, it's an error if there are no selectors that contain `.notice`. It's also an error if the only selector containing `.notice` is `h1.notice`, since `h1` conflicts with `a` and so no new selector would be generated.

Sometimes, though, you want to allow an `@extend` not to produce any new selectors. To do so, just add the `!optional` flag after the selector. For example:

```
a.important {
  @extend .notice !optional;
}
```

`@extend` in Directives

There are some restrictions on the use of `@extend` within directives such as `@media`. Sass is unable to make CSS rules outside of the `@media` block apply to selectors inside it without creating a huge amount of stylesheet bloat by copying styles all over the place. This means that if you use `@extend` within `@media` (or other CSS directives), you may only extend selectors that appear within the same directive block.

For example, the following works fine:

```
@media print {
  .error {
    border: 1px #f00;
    background-color: #fdd;
  }
  .seriousError {
    @extend .error;
    border-width: 3px;
  }
}
```

But this is an error:

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}

@media print {
  .seriousError {
    // INVALID EXTEND: .error is used outside of the "@media print" directive
    @extend .error;
    border-width: 3px;
  }
}
```

Someday we hope to have `@extend` supported natively in the browser, which will allow it to be used within `@media` and other directives.

`@at-root`

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. It can either be used with a single inline selector:

```
.parent {
  ...
  @at-root .child { ... }
}
```

Which would produce:

```
.parent { ... }
.child { ... }
```

Or it can be used with a block containing multiple selectors:

```
.parent {
  ...
  @at-root {
    .child1 { ... }
    .child2 { ... }
  }
  .step-child { ... }
}
```

Which would output the following:

```
.parent { ... }
.child1 { ... }
.child2 { ... }
.parent .step-child { ... }
```

`@at-root (without: ...)` and `@at-root (with: ...)`

By default, `@at-root` just excludes selectors. However, it's also possible to use `@at-root` to move outside of nested directives such as `@media` as well. For example:

```
@media print {
  .page {
    width: 8in;
    @at-root (without: media) {
      color: red;
    }
  }
}
```

produces:

```
@media print {
  .page {
    width: 8in;
  }
}
.page {
  color: red;
}
```

You can use `@at-root (without: ...)` to move outside of any directive. You can also do it with multiple directives separated by a space: `@at-root (without: media supports)` moves outside of both `@media` and `@supports` queries.

There are two special values you can pass to `@at-root`. "rule" refers to normal CSS rules; `@at-root (without: rule)` is the same as `@at-root` with no query. `@at-root (without: all)` means that the styles should be moved outside of *all* directives and CSS rules.

If you want to specify which directives or rules to include, rather than listing which ones should be excluded, you can use `with` instead of `without`. For example, `@at-root (with: rule)` will move outside of all directives, but will preserve any CSS rules.

`@debug`

The `@debug` directive prints the value of a SassScript expression to the standard error output stream. It's useful for debugging Sass files that have complicated SassScript going on. For example:

```
@debug 10em + 12em;
```

outputs:

```
Line 1 DEBUG: 22em
```

`@warn`

The `@warn` directive prints the value of a SassScript expression to the standard error output stream. It's useful for libraries that need to warn users of deprecations or recovering from minor mixin usage mistakes. There are two major distinctions between `@warn` and `@debug`:

1. You can turn warnings off with the `--quiet` command-line option or the `:quiet` Sass option.
2. A stylesheet trace will be printed out along with the message so that the user being warned can see where their styles caused the warning.

Usage Example:

```
@mixin adjust-location($x, $y) {
  @if unitless($x) {
    @warn "Assuming #{$x} to be in pixels";
    $x: 1px * $x;
  }
  @if unitless($y) {
    @warn "Assuming #{$y} to be in pixels";
    $y: 1px * $y;
  }
  position: relative; left: $x; top: $y;
}
```

`@error`

The `@error` directive throws the value of a SassScript expression as a fatal error, including a nice stack trace. It's useful for validating arguments to mixins and functions. For example:

```
@mixin adjust-location($x, $y) {
  @if unitless($x) {
    @error "$x may not be unitless, was #{$x}.";
  }
  @if unitless($y) {
    @error "$y may not be unitless, was #{$y}.";
  }
  position: relative; left: $x; top: $y;
}
```

There is currently no way to catch errors.

Control Directives & Expressions

SassScript supports basic control directives and expressions for including styles only under some conditions or including the same style several times with variations.

Note: Control directives are an advanced feature, and are uncommon in day-to-day styling. They exist mainly for use in mixins, particularly those that are part of libraries like Compass, and so require substantial flexibility.

`if()`

The built-in `if()` function allows you to branch on a condition and returns only one of two possible outcomes. It can be used in any script context. The `if` function only evaluates the argument corresponding to the one that it will return – this allows you to refer to variables that may not be defined or to have calculations that would otherwise cause an error (E.g. divide by zero).

`@if`

The `@if` directive takes a SassScript expression and uses the styles nested beneath it if the expression returns anything other than `false` or `null`:

```
p {
  @if 1 + 1 == 2 { border: 1px solid;  }
  @if 5 < 3      { border: 2px dotted; }
  @if null       { border: 3px double; }
}
```

is compiled to:

```
p {
  border: 1px solid; }
```

The `@if` statement can be followed by several `@else if` statements and one `@else` statement. If the `@if` statement fails, the `@else if` statements are tried in order until one succeeds or the `@else` is reached. For example:

```
$type: monster;
p {
  @if $type == ocean {
    color: blue;
  } @else if $type == matador {
    color: red;
  } @else if $type == monster {
    color: green;
  } @else {
    color: black;
  }
}
```

is compiled to:

```
p {
  color: green; }
```

`@for`

The `@for` directive repeatedly outputs a set of styles. For each repetition, a counter variable is used to adjust the output. The directive has two forms: `@for $var from <start> through <end>` and `@for $var from <start> to <end>`. Note the difference in the keywords `through` and `to`. `$var` can be any variable name, like `$i`; `<start>` and `<end>` are SassScript expressions that should return integers. When `<start>` is greater than `<end>` the counter will decrement instead of increment.

The `@for` statement sets `$var` to each successive number in the specified range and each time outputs the nested styles using that value of `$var`. For the form `from ... through`, the range *includes* the values of `<start>` and `<end>`, but the form `from ... to` runs up to *but not including* the value of `<end>`. Using the `through` syntax,

```
@for $i from 1 through 3 {
  .item-#{$i} { width: 2em * $i; }
}
```

is compiled to:

```
.item-1 {
  width: 2em; }
.item-2 {
  width: 4em; }
.item-3 {
  width: 6em; }
```

`@each`

The `@each` directive usually has the form `@each $var in <list or map>`. `$var` can be any variable name, like `$length` or `$name`, and `<list or map>` is a SassScript expression that returns a list or a map.

The `@each` rule sets `$var` to each item in the list or map, then outputs the styles it contains using that value of `$var`. For example:

```
@each $animal in puma, sea-slug, egret, salamander {
  .#{$animal}-icon {
    background-image: url('/images/#{$animal}.png');
  }
}
```

is compiled to:

```
.puma-icon {
  background-image: url('/images/puma.png'); }
```

```
.sea-slug-icon {
  background-image: url('/images/sea-slug.png'); }
.egret-icon {
  background-image: url('/images/egret.png'); }
.salamander-icon {
  background-image: url('/images/salamander.png'); }
```

Multiple Assignment

The `@each` directive can also use multiple variables, as in `@each $var1, $var2, ... in <list>`. If `<list>` is a list of lists, each element of the sub-lists is assigned to the respective variable. For example:

```
@each $animal, $color, $cursor in (puma, black, default),
                                  (sea-slug, blue, pointer),
                                  (egret, white, move) {
  .#{$animal}-icon {
    background-image: url('/images/#{$animal}.png');
    border: 2px solid $color;
    cursor: $cursor;
  }
}
```

is compiled to:

```
.puma-icon {
  background-image: url('/images/puma.png');
  border: 2px solid black;
  cursor: default; }
.sea-slug-icon {
  background-image: url('/images/sea-slug.png');
  border: 2px solid blue;
  cursor: pointer; }
.egret-icon {
  background-image: url('/images/egret.png');
  border: 2px solid white;
  cursor: move; }
```

Since maps are treated as lists of pairs, multiple assignment works with them as well. For example:

```
@each $header, $size in (h1: 2em, h2: 1.5em, h3: 1.2em) {
  #{$header} {
    font-size: $size;
  }
}
```

is compiled to:

```
h1 {
  font-size: 2em; }
h2 {
  font-size: 1.5em; }
h3 {
  font-size: 1.2em; }
```

`@while`

The `@while` directive takes a SassScript expression and repeatedly outputs the nested styles until the statement evaluates to `false`. This can be used to achieve more complex looping than the `@for` statement is capable of, although this is rarely necessary. For example:

```
$i: 6;
@while $i > 0 {
  .item-#{$i} { width: 2em * $i; }
  $i: $i - 2;
}
```

is compiled to:

```
.item-6 {
  width: 12em; }

.item-4 {
  width: 8em; }

.item-2 {
  width: 4em; }
```

Mixin Directives

Mixins allow you to define styles that can be re-used throughout the stylesheet without needing to resort to non-semantic classes like `.float-left`. Mixins can also contain full CSS rules, and anything else allowed elsewhere in a Sass document. They can even take arguments which allows you to produce a wide variety of styles with very few mixins.

Defining a Mixin: `@mixin`

Mixins are defined with the `@mixin` directive. It's followed by the name of the mixin and optionally the arguments, and a block containing the contents of the mixin. For example, the `large-text` mixin is defined as follows:

```
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}
```

Mixins may also contain selectors, possibly mixed with properties. The selectors can even contain parent references. For example:

```
@mixin clearfix {
  display: inline-block;
  &:after {
    content: ".";
    display: block;
    height: 0;
    clear: both;
    visibility: hidden;
  }
  * html & { height: 1px }
}
```

Including a Mixin: `@include`

Mixins are included in the document with the `@include` directive. This takes the name of a mixin and optionally arguments to pass to it, and includes the styles defined by that mixin into the current rule. For example:

```
.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

is compiled to:

```
.page-title {
  font-family: Arial;
  font-size: 20px;
  font-weight: bold;
  color: #ff0000;
  padding: 4px;
  margin-top: 10px; }
```

Mixins may also be included outside of any rule (that is, at the root of the document) as long as they don't directly define any properties or use any parent references. For example:

```
@mixin silly-links {
  a {
    color: blue;
    background-color: red;
  }
}

@include silly-links;
```

is compiled to:

```
a {
  color: blue;
  background-color: red; }
```

Mixin definitions can also include other mixins. For example:

```
@mixin compound {
  @include highlighted-background;
  @include header-text;
}

@mixin highlighted-background { background-color: #fc0; }
@mixin header-text { font-size: 20px; }
```

Mixins may include themselves. This is different than the behavior of Sass versions prior to 3.3, where mixin recursion was forbidden.

Mixins that only define descendent selectors can be safely mixed into the top most level of a document.

Arguments

Mixins can take SassScript values as arguments, which are given when the mixin is included and made available within the mixin as variables.

When defining a mixin, the arguments are written as variable names separated by commas, all in parentheses after the name. Then when including the mixin, values can be passed in in the same manner. For example:

```
@mixin sexy-border($color, $width) {
  border: {
    color: $color;
    width: $width;
    style: dashed;
  }
}

p { @include sexy-border(blue, 1in); }
```

is compiled to:

```
p {
  border-color: blue;
  border-width: 1in;
  border-style: dashed; }
```

Mixins can also specify default values for their arguments using the normal variable-setting syntax. Then when the mixin is included, if it doesn't pass in that argument, the default value will be used instead. For example:

```
@mixin sexy-border($color, $width: 1in) {
  border: {
    color: $color;
    width: $width;
    style: dashed;
  }
}
p { @include sexy-border(blue); }
h1 { @include sexy-border(blue, 2in); }
```

is compiled to:

```
p {
  border-color: blue;
  border-width: 1in;
  border-style: dashed; }

h1 {
  border-color: blue;
  border-width: 2in;
  border-style: dashed; }
```

Keyword Arguments

Mixins can also be included using explicit keyword arguments. For instance, the above example could be written as:

```
p { @include sexy-border($color: blue); }
h1 { @include sexy-border($color: blue, $width: 2in); }
```

While this is less concise, it can make the stylesheet easier to read. It also allows functions to present more flexible interfaces, providing many arguments without becoming difficult to call.

Named arguments can be passed in any order, and arguments with default values can be omitted. Since the named arguments are variable names, underscores and dashes can be used interchangeably.

Variable Arguments

Sometimes it makes sense for a mixin or function to take an unknown number of arguments. For example, a mixin for creating box shadows might take any number of shadows as arguments. For these situations, Sass supports “variable arguments,” which are arguments at the end of a mixin or function declaration that take all leftover arguments and package them up as a list. These arguments look just like normal arguments, but are followed by `...`. For example:

```
@mixin box-shadow($shadows...) {
  -moz-box-shadow: $shadows;
  -webkit-box-shadow: $shadows;
  box-shadow: $shadows;
}

.shadows {
  @include box-shadow(0px 4px 5px #666, 2px 6px 10px #999);
}
```

is compiled to:

```
.shadows {
  -moz-box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
  -webkit-box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
  box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
}
```

Variable arguments also contain any keyword arguments passed to the mixin or function. These can be accessed using the `keywords($args)` function, which returns them as a map from strings (without `$`) to values.

Variable arguments can also be used when calling a mixin. Using the same syntax, you can expand a list of values so that each value is passed as a separate argument, or expand a map of values so that each pair is treated as a keyword argument. For example:

```
@mixin colors($text, $background, $border) {
  color: $text;
  background-color: $background;
  border-color: $border;
}
```

```
$values: #ff0000, #00ff00, #0000ff;
.primary {
  @include colors($values...);
}

$value-map: (text: #00ff00, background: #0000ff, border: #ff0000);
.secondary {
  @include colors($value-map...);
}
```

is compiled to:

```
.primary {
  color: #ff0000;
  background-color: #00ff00;
  border-color: #0000ff;
}

.secondary {
  color: #00ff00;
  background-color: #0000ff;
  border-color: #ff0000;
}
```

You can pass both an argument list and a map as long as the list comes before the map, as in `@include colors($values..., $map...)`.

You can use variable arguments to wrap a mixin and add additional styles without changing the argument signature of the mixin. If you do, keyword arguments will get directly passed through to the wrapped mixin. For example:

```
@mixin wrapped-stylish-mixin($args...) {
  font-weight: bold;
  @include stylish-mixin($args...);
}

.stylish {
  // The $width argument will get passed on to "stylish-mixin" as a keyword
  @include wrapped-stylish-mixin(#00ff00, $width: 100px);
}
```

Passing Content Blocks to a Mixin

It is possible to pass a block of styles to the mixin for placement within the styles included by the mixin. The styles will appear at the location of any `@content` directives found within the mixin. This makes it possible to define abstractions relating to the construction of selectors and directives.

For example:

```
@mixin apply-to-ie6-only {
  * html {
    @content;
  }
}
@include apply-to-ie6-only {
  #logo {
    background-image: url(/logo.gif);
  }
}
```

Generates:

```
* html #logo {
  background-image: url(/logo.gif);
}
```

The same mixins can be done in the `.sass` shorthand syntax:

```
=apply-to-ie6-only
```

```
  * html
    @content

+apply-to-ie6-only
  #logo
    background-image: url(/logo.gif)
```

Note: when the `@content` directive is specified more than once or in a loop, the style block will be duplicated with each invocation.

Variable Scope and Content Blocks

The block of content passed to a mixin are evaluated in the scope where the block is defined, not in the scope of the mixin. This means that variables local to the mixin **cannot** be used within the passed style block and variables will resolve to the global value:

```
$color: white;
@mixin colors($color: blue) {
  background-color: $color;
  @content;
  border-color: $color;
}
.colors {
  @include colors { color: $color; }
}
```

Compiles to:

```
.colors {
  background-color: blue;
  color: white;
  border-color: blue;
}
```

Additionally, this makes it clear that the variables and mixins that are used within the passed block are related to the other styles around where the block is defined. For example:

```
#sidebar {
  $sidebar-width: 300px;
  width: $sidebar-width;
  @include smartphone {
    width: $sidebar-width / 3;
  }
}
```

Function Directives

It is possible to define your own functions in sass and use them in any value or script context. For example:

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

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

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

Becomes:

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

As you can see functions can access any globally defined variables as well as accept arguments just like a mixin. A function may have several statements contained within it, and you must call `@return` to set the return value of the function.

As with mixins, you can call Sass-defined functions using keyword arguments. In the above example we could have called the function like this:

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

It is recommended that you prefix your functions to avoid naming conflicts and so that readers of your stylesheets know they are not part of Sass or CSS. For example, if you work for ACME Corp, you might have named the function above `-acme-grid-width`.

User-defined functions also support variable arguments in the same way as mixins.

Output Style

Although the default CSS style that Sass outputs is very nice and reflects the structure of the document, tastes and needs vary and so Sass supports several other styles.

Sass allows you to choose between four different output styles by setting the `:style` option or using the `--style` command-line flag.

`:nested`

Nested style is the default Sass style, because it reflects the structure of the CSS styles and the HTML document they're styling. Each property has its own line, but the indentation isn't constant. Each rule is indented based on how deeply it's nested. For example:

```
#main {
  color: #fff;
  background-color: #000; }
  #main p {
    width: 10em; }

.huge {
  font-size: 10em;
  font-weight: bold;
  text-decoration: underline; }
```

Nested style is very useful when looking at large CSS files: it allows you to easily grasp the structure of the file without actually reading anything.

`:expanded`

Expanded is a more typical human-made CSS style, with each property and rule taking up one line. Properties are indented within the rules, but the rules aren't indented in any special way. For example:

```
#main {
  color: #fff;
  background-color: #000;
}
#main p {
  width: 10em;
}

.huge {
  font-size: 10em;
  font-weight: bold;
  text-decoration: underline;
}
```

`:compact`

Compact style takes up less space than Nested or Expanded. It also draws the focus more to the selectors than to their properties. Each CSS rule takes up only one line, with every property defined on that line. Nested rules are placed next to each other with no newline, while separate groups of rules have newlines between them. For example:

```
#main { color: #fff; background-color: #000; }
#main p { width: 10em; }

.huge { font-size: 10em; font-weight: bold; text-decoration: underline; }
```

`:compressed`

Compressed style takes up the minimum amount of space possible, having no whitespace except that necessary to separate selectors and a newline at the end of the file. It also includes some other minor compressions, such as choosing the smallest representation for colors. It's not meant to be humanreadable. For example:

```
#main{color:#fff;background-color:#000}#main p{width:10em}.huge{font-size:10em;font-weight:bold;text-decoration:underline}
```

Extending Sass

Sass provides a number of advanced customizations for users with unique requirements. Using these features requires a strong understanding of Ruby.

Defining Custom Sass Functions

Users can define their own Sass functions using the Ruby API. For more information, see the source documentation.

Cache Stores

Sass caches parsed documents so that they can be reused without parsing them again unless they have changed. By default, Sass will write these cache files to a location on the filesystem indicated by `:cache_location`. If you cannot write to the filesystem or need to share cache across ruby processes or machines, then you can define your own cache store and set the `:cache_store` option. For details on creating your own cache store, please see the source documentation.

Custom Importers

Sass importers are in charge of taking paths passed to `@import` and finding the appropriate Sass code for those paths. By default, this code is loaded from the filesystem, but importers could be added to load from a database, over HTTP, or use a different file naming scheme than what Sass expects.

Each importer is in charge of a single load path (or whatever the corresponding notion is for the backend). Importers can be placed in the `:load_paths` array alongside normal filesystem paths.

When resolving an `@import`, Sass will go through the load paths looking for an importer that successfully imports the path. Once one is found, the imported file is used.

User-created importers must inherit from Sass::Importers::Base.