

CPSC 448 Final Report

Student: Christopher Powroznik (44281146)

Faculty Supervisor: Elisa Baniassad

Table of Contents

[Overview 2](#_Toc21192986)

[Learning Objectives 2](#_Toc21192987)

[Survey of Prior Work 2](#_Toc21192988)

[The Existing Colour Patterns 2](#_Toc21192989)

[Compliments 2](#_Toc21192990)

[Analogous 2](#_Toc21192991)

[Triadic 2](#_Toc21192992)

[Tetradic 2](#_Toc21192993)

[Primary Secondary and Tertiary Schemes 2](#_Toc21192994)

[The ‘S’ Pattern 3](#_Toc21192995)

[Dave Green's ‘cubehelix’ Colour Scheme 3](#_Toc21192996)

[Shape Based Selection 3](#_Toc21192997)

[What is Colour Chisel? 3](#_Toc21192998)

[Milestones 3](#_Toc21192999)

[Create an EBNF 3](#_Toc21193000)

[Reflection 4](#_Toc21193001)

[Initialize Project Structure and Test Suite. 4](#_Toc21193002)

[Reflection 5](#_Toc21193003)

[Create Interface for Library and Use it to Create Tests. 6](#_Toc21193004)

[Make Compilable File to JavaScript 7](#_Toc21193005)

[Make Compilable File to SCSS 7](#_Toc21193006)

[Get Feedback and Adjust Library a 2nd Time 7](#_Toc21193007)

[Make Online Tool to Create JS and SCSS Exports Without a Dependency 7](#_Toc21193008)

[Release as NPM Module 7](#_Toc21193009)

[Bibliography 7](#_Toc21193010)

# Overview

The following is a document outlying the learning objectives, development and overall progression Christopher Powroznik CPSC 448 project ‘Colour Chisel’.

# Learning Objectives

As agreed between the Elisa Baniassad and Christopher Powroznik. The following learning objectives are the goals for the project. By the end of the project Christopher will have covered all of the following…

* Learn how to develop a continuous integration setup involving automated testing, publishing and deployment of a web demo as a hook from committing code.
* Learn to integrate a compilation tool into development pipelines such as web pack and bower
* Learn geometric transformations in a circular space programmatically to do operations on the colour wheel
* Learn to setup and develop a project without a boilerplate
* Learn to create procedural usage libraries that recursively repeats functionality
* Explore different colouring techniques programmatically as opposed to traditional primary, secondary and tertiary design
* Deepen my learning to iteratively make code design choices and refactor code based on feedback and direction of the project.
* Deepen my learning of developing test suites to perform unit testing, end-to-end testing and black box testing of my code

# Survey of Prior Work

## The Existing Colour Patterns

It is quick to find that there are defined ways to finding colours that look good together, which are even mentioned in Isaac Newton’s initial paper on the colour spectrum (Newton, 1665). All of these have inspired the default actions in my Colour Chisel library.

### Compliments

Compliment colours are colours that are directly across from each other on the colour wheel. They contrast entirely with and appear more prevalent when next to each other.

### Analogous

An analogous colour is one that is a rotation from one colour to another that is usually just to the left or right of the first colour. These are commonly used as accents, in order to highlight slight changes and create a deeper colour, by including a left or right analogous of a primary colour.

### Triadic

### Tetradic

## Primary Secondary and Tertiary Schemes

## The ‘S’ Pattern

## Dave Green's ‘cubehelix’ Colour Scheme

# Shape Based Selection

# What is Colour Chisel?

# Milestones

## Create an EBNF

(September 19, 2019 🡪 September 24, 2019)

I started out by trying to find the most standard way to produce an EBNF. I came across the article ‘the language of languages’ (Might, n.d.). It covered writing an EBNF in a very robust methodology. I familiarized myself with the syntax of an EBNF by following examples from Wikipedia (Extended Backus–Naur form , n.d.). I began writing my EBNF and checking my work using ‘Railroad Diagram Generator’ (Rademacher, 2019). I first created a base EBNF, that was very barebones, especially with pathing. I wasn’t sure on how to handle variables and getting certain colours yet.

expression  
 ::= (const | path) ("," manipulation)\*  
  
/\* Pathing \*/  
path ::= "[" (expression) ("," expression)\* "]"  
  
/\* Manipulations \*/  
manipulation  
 ::= analogous  
 | compliment  
analogous  
 ::= (A|a) range?  
compliment  
 ::= C  
 | c  
range ::= '-'? [0-9]+ ("." [0-9]+)?  
  
/\* Constants \*/  
const ::= '(' ( rgb | rgba ) ')'  
rgba ::= 'rgba(' colourValue ',' colourValue ',' colourValue ',' colourValue ')'  
rgb ::= 'rgb(' colourValue ',' colourValue ',' colourValue ')'  
hex ::= '#'? hexit hexit hexit ( hexit hexit )?  
hexit ::= [0-9a-fA-F]  
colourValue  
 ::= ( [1-9] | '1' [0-9] | '2' [0-4] )? [0-9]  
 | '25' [0-5]  
rgba ::= 'rgba(' colourValue ',' colourValue ',' colourValue ',' colourValue ')'

After reflecting and reading the article \_\_\_\_\_. I decided to edit my original definition of expression to handle the case of a const or path being a saved variable. This led me to the new EBNF shown below. I handled paths in a similar way to JavaScript arrays. Where defining a path, I could reference a spot in the path by calling the path or variable and indicating the index of the colour I wanted to take from the path.

expression  
 ::= (const | path | variableName) ("," manipulation)\*  
  
/\* Variables \*/  
variable ::= var variableName "=" (expression | path)  
pathVariable ::= variableName "[" [0-9]+ "]"  
variableName ::= [a-fA-F]+  
  
/\* Manipulations \*/  
manipulation  
 ::= analogous  
 | compliment  
analogous  
 ::= (A|a) range?  
compliment  
 ::= C  
 | c  
range ::= '-'? [0-9]+ ("." [0-9]+)?  
  
/\* Constants \*/  
const ::= '(' ( rgb | rgba ) ')'  
rgba ::= 'rgba(' colourValue ',' colourValue ',' colourValue ',' colourValue ')'  
rgb ::= 'rgb(' colourValue ',' colourValue ',' colourValue ')'  
hex ::= '#'? hexit hexit hexit ( hexit hexit )?  
hexit ::= [0-9a-fA-F]  
colourValue  
 ::= ( [1-9] | '1' [0-9] | '2' [0-4] )? [0-9]  
 | '25' [0-5]  
rgba ::= 'rgba(' colourValue ',' colourValue ',' colourValue ',' colourValue ')'  
  
/\* Pathing \*/  
path ::= "[" (expression) ("," expression)\* "]"  
rotate ::= expression ".rotate(" range ")"  
translate  
 ::= expression ".translate(" colourValue "," colourValue ")"  
scale ::= expression ".scale(" colourValue ")"

### Reflection

Creating the EBNF was more time consuming than I had believed, my biggest fault was trying to define a scheme, before determining how I would even go about getting data from a path. When I started to think about using the JavaScript interface, rather than the language interface, I had an easier time. In the future, I would try and model my language after I determine all features to be included.

## Initialize Project Structure and Test Suite.

(September 24, 2019 🡪 October 5, 2019)

The first thing I needed to discover in this phase was to decide on the public appeal of the project. I intend on releasing this library publicly on npm and GitHub, thus I wanted to follow a standard. I also do a lot of my work out of the repository and wanted to keep it this way. I found there were common traits to naming directories and organization after looking through a handful of other npm modules including moment[[1]](#footnote-1), lodash[[2]](#footnote-2) and react[[3]](#footnote-3). It is common for npm packages to have a series of committed and non-committed files. The repository will hold the ‘src’ directory, which tends to be just the development code. Then there are usually 2 other directories, that are made upon compilation; these are ‘lib’ and ‘dist’. While both seemed to be usually interchangeably in some repositories. It is common to have ‘dist’ as the compiled code, while the ‘lib’ contains the minified version of the compiled code.

My project is being built in Typescript, so I decided to have an ‘src’ for the Typescript code and ‘dist’ for the compiled JS code. However, I decided against putting a ‘lib’ with a minification step. These reasons are the scope of my project. ‘lib’ directories are often seen as optional and only necessary for large libraries that would benefit from a minification. My project is tiny in comparison, and the benefits of minification are negligible. It is possible in the future if scope expands to include a minification to the release schedule.

At the end of this step my directories were set up as…



For the test suite, I also looked at the most popular options in order to stay uniform. Coincidence enough, I landed on using Mocha[[4]](#footnote-4) and Chai[[5]](#footnote-5). I installed these quite easily and left the first initial tests for the next milestone.

Lastly, I wanted a basic setup for continuous testing and deploying. I did this via the new Github Actions[[6]](#footnote-6). Since this was a very new service at the time of writing this, I used the boilerplate and tutorial provided by Github for node projects. After following the tutorial, I managed to run the building and testing of my project with every commit on the project. The deployment will be saved for the later milestone.

name: Node CI  
  
on: [push]  
  
jobs:  
 build:  
  
 runs-on: ubuntu-latest  
  
 strategy:  
 matrix:  
 node-version: [8.x, 10.x, 12.x]  
  
 steps:  
 - uses: actions/checkout@v1  
 - name: Use Node.js ${{ matrix.node-version }}  
 uses: actions/setup-node@v1  
 with:  
 node-version: ${{ matrix.node-version }}  
 - name: npm install, build, and test  
 run: |  
 npm install  
 npm run build --if-present  
 npm test  
 env:  
 CI: true

### Reflection

This section I was the most worried about. I felt like there were so many examples to look at, but there was no exact idea of a standard. Each project is different, and there are so many ways to go about project structure. I found myself looking every popular module on npm and taking notes on what the best way to organize is.

I had one unique case of having files for this report being stored in the repository, which I wanted to keep as I believe they provide context to my work and are valuable to anyone using my library. I came upon the idea of adding a ‘meta’ directory. This was my own idea to keep things outside of the exact scope of the project but would be interesting to have in the repository.

## Create Interface for Library and Use it to Create Tests.

(September 24, 2019 🡪 October 5, 2019)

I studied the earlier EBNF and momentJS declarations, to come up with an effective way to design a procedural interface. I landed creating 2 primary interfaces which are

At first, I had a very long test suite, but found I could just programmatically generate the tests. I made factory as so…

describe("test the basic utilities of passing a colour in and exporting to different variations", () => {  
 const hex = '#646464';  
 const rgb = 'rgb(100,100,100)';  
 const rgba = 'rgba(100,100,100,1)';  
  
 describe("use hex as input", testInput(hex));  
 describe("use rgb as input", testInput(hex));  
 describe("use rgba as input", testInput(hex));  
  
 function testInput(input: string) {  
 return () => {  
 const cc = colourChisel(input);  
 it("output to hex", () => {  
 expect(cc.hex()).to.be.equal(hex);  
 });  
  
 it("output to rgb", () => {  
 expect(cc.rgb()).to.be.equal(rgb);  
 });  
  
 it("output to rgba", () => {  
 expect(cc.rgba()).to.be.equal(rgba)  
 });  
 }  
 }  
});

I used this functionality with further unit tests to more effectively cover all edge cases through iteration.

To make the test suite easier to write, I wrote a file on constants around the ‘S’ pattern using Chroma-js. Chroma-js allowed me to do simple operations on single points of the colour wheel.

import chroma from "chroma-js";  
  
// Standard 'S'  
export const s0 = chroma("red").hex();  
export const s1 = chroma(s0).set("hsl.h", "-45").hex();  
export const s2 = chroma(s1).set("hsl.h", "-45").hex();  
export const s3 = chroma(s0).set("hsl.h", "+180").hex();  
export const s4 = chroma(s1).set("hsl.h", "+180").hex();  
export const s5 = chroma(s2).set("hsl.h", "+180").hex();  
export const sPath = [s0, s1, s2, s3, s4, s5];  
  
// Rotated 'S' 90 degrees  
export const sr90Path = sPath.map(h => chroma(h).set("hsl.h", "+90").hex());  
  
// Rotated 'S' -90 degrees  
export const srNeg90Path = sPath.map(h => chroma(h).set("hsl.h", "-90").hex());  
  
// Scaled 'S' 0.5x  
export const sScaleHalfPath = sPath.map(h => chroma(h).set("hsl.s", Math.round(chroma(h).hsl[1] \* 0.5)));  
  
// Scaled 'S' 2x  
export const sScaleDoublePath = sPath.map(h => chroma(h).set("hsl.s", Math.round(chroma(h).hsl[1] \* 2)));

Extend Library for Procedural Usage

(September 24, 2019 🡪 October 5, 2019)

I decided to change the order of 2 of my milestones for a smoother implementation. I switched ‘Extend Library for Procedural Usage’ and ‘Implement Language Directly into Runtime Compiler’ because I feel that the evaluation of my language will reply upon my JavaScript functionality. This is different than the initial proposal, where these two were reversed. After reflecting on the creating of the interfaces, it is clear this is the more optimal order.

Implement Language Directly into Runtime Compiler

Get Feedback and Adjust Library

## Make Compilable File to JavaScript

## Make Compilable File to SCSS

## Get Feedback and Adjust Library a 2nd Time

## Make Online Tool to Create JS and SCSS Exports Without a Dependency

## Release as NPM Module

# Bibliography

*Extended Backus–Naur form* . (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Extended\_Backus%E2%80%93Naur\_form

Might, M. (n.d.). *the language of languages*. Retrieved from mattmight.net: http://matt.might.net

Newton, I. (1665). Newton’s Color Theory.

Rademacher, G. (2019, Augst 19). *Railroad Diagram Generator*. Retrieved from bottlecaps: https://bottlecaps.de/rr/ui

1. https://github.com/moment/moment [↑](#footnote-ref-1)
2. https://github.com/lodash/lodash [↑](#footnote-ref-2)
3. https://github.com/facebook/react [↑](#footnote-ref-3)
4. https://mochajs.org/ [↑](#footnote-ref-4)
5. https://www.chaijs.com/ [↑](#footnote-ref-5)
6. https://github.com/features/actions [↑](#footnote-ref-6)