5/20/2018

Source Code Analyser Engine

Technical Guide

Bernard O'Connor
14367821
Dr David Sinclair

TABLE OF CONTENTS

OVERVIEW

The project is the Source Code Analyser Engine. The aim of the project is to create a library that can be used to promote the writing of quality code, that other developers may use to asses their own code or to incorporate into their own projects.

The project is split into two sub projects / components; the Library and the Web Application.

The Library is the main focus of the project. The library is what analyses the source code and returns either style suggestions or the generated Abstract Syntax Tree.

The Web Application acts as both the home of the project and as an example of the types of applications a user of the Library could make. The Web Application offers a number of services that's allow the user to try out the library before installing it locally.

GLOSSARY

- **Node Based Styling:** My term for applying Style Rules to Nodes on my Abstract Syntax Tree
- **Library:** A collection of software components that can be imported and used by others
- **Clojure / ClojureScript**: A functional language(s) build upon the JVM and JavaScript respectively
- **JVM**: Java Virtual Machine
- **Quote in Clojure**: Tells the Clojure Reader/Evaluator not to evaluate a symbol
- **Frontend**: The presentation layer of a piece of software. Can be thought of as the browser client
- **Backend**: The data access layer of a piece of software. Can be thought of as the web-server
- **Style Rulebooks**: JSON files used by my library that defines how to build an AST
- **Grammar**: Describes the form/structure of a language (in this case programming languages)
- **Abstract Syntax Tree (AST):** A tree representation of the structure of a piece of source code
- **Tokenisation**: The process of turning a sequence of characters into tokens.
- **Tokens**: A set of characters grouped together.
- **Production Rules**: Rules that describe how a node is formed.
- **Node**: An entry to the Abstract Syntax Tree.
- **Epsilon Transitions**: A special grammar transition that does not consume any tokens.
- **Regex**: Regular Expressions- powerful method of selecting certain portions of strings.
- **Vector**: An ordered data structure like Arrays in other languages.
- **Forward Declaration**: Tells the compiler that a function will be declared later.
- **Namespace**: A collection of functions in a functional language. Vaguely similar to classes.
- **Functional Language**: Type of programming language that treats computation as evaluation of mathematical functions and avoids the use of mutable data where possible.
- **Mutable data**: Data that can be altered at a later date.
- **Collection**: Here used to describe Clojure data structures that are immutable and persistent.
- **Lexical Analysis:** The process of tokenizing source code and stripping out unwanted tokens
- **Leiningen:** A project build manager for Clojure/ClojureScript Projects
- **Lazy-Sequence:** A Clojure data structure whose contents are only evaluated / computed when called upon.
- **Uberjar:** A jar file that contains all its dependencies within it
- **AWS**: Amazon Web Services – A cloud computing platform.

ARCHITECTURAL DESIGN

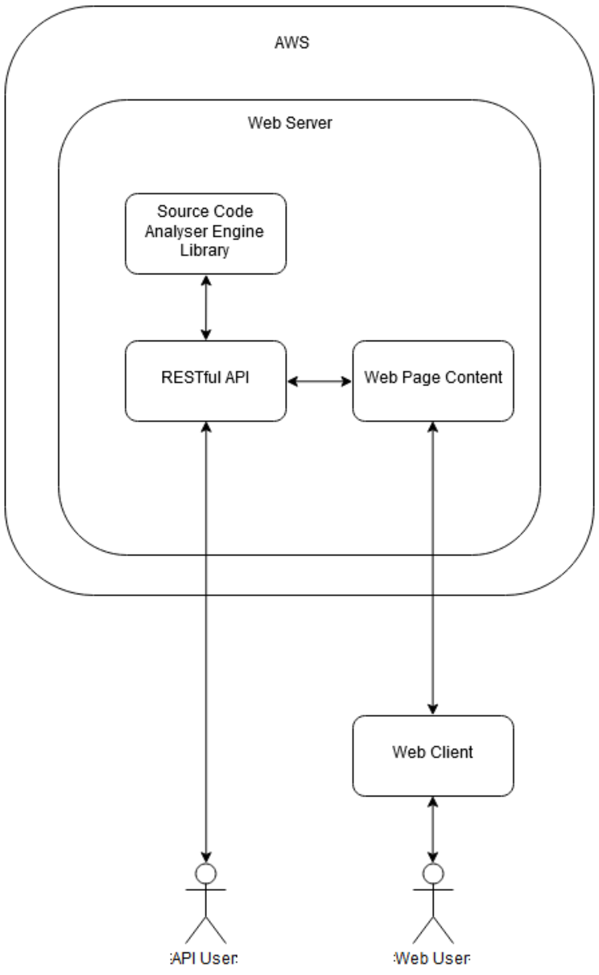
Below we talk about the design of the System as a whole, and then delve deeper into the design of the Library and Web Application.

HIGH LEVEL DESIGN

As stated previously, the Project is split into two major components, the Library and the Web Application that incorporates the library. The Web Application is currently deployed to an Ubuntu Server on AWS, providing high availability and ease of management.

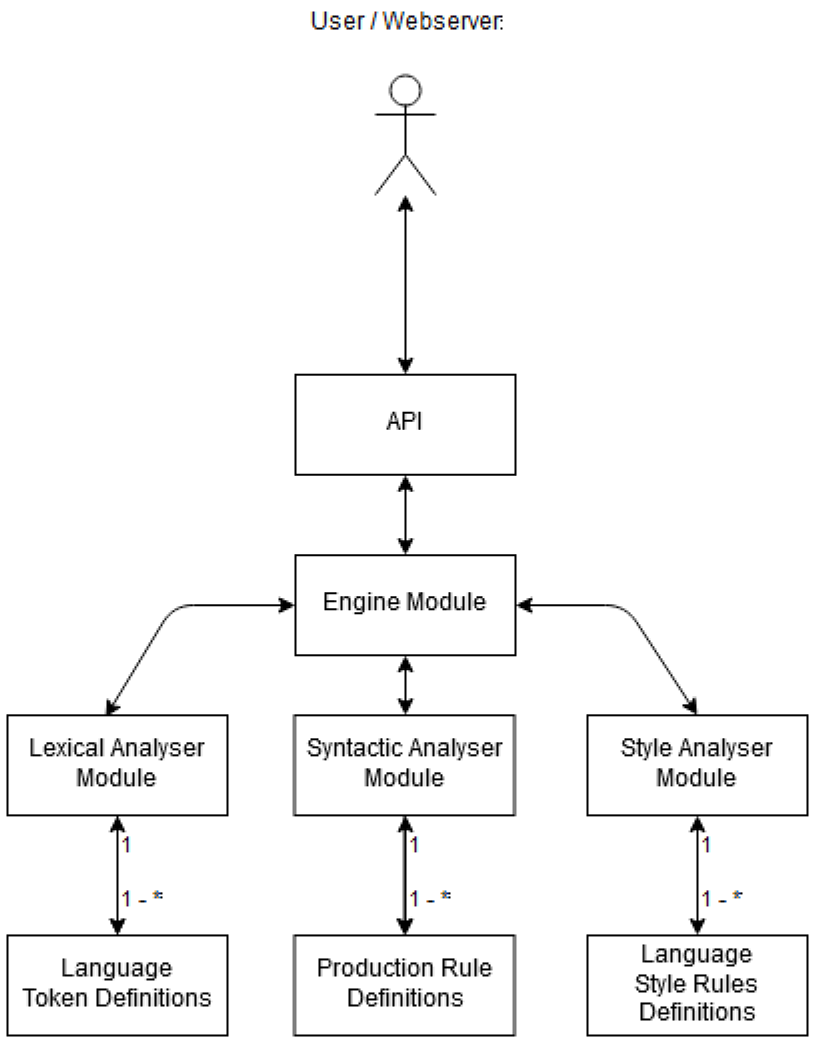
On the Ubuntu Server the Web Application is running on a Jetty Server. From here we can serve out either Web Page Content to the Web Clients or lets the user to access the RESTful API. The RESTful API in a request and calls the Source Code Analyser Engine to analyse a piece of source code along with a Style Rulebook and return a result.

The Style Rulebook can be written by the user to define the making of an Abstract Syntax Tree and the definition of Style Rules for any language, giving the project it's modular nature



LIBRARY: APPLICATION ARCHITECTURE

The library is responsible for parsing the request of a user, containing a piece of source code and a Style Rulebook, and then returning either style suggestions or an Abstract Syntax Tree. See the section "**Writing Rulebooks – Basic Format**" in the **User Manual** for more in depth information on the Style Rulebook Format.


User / Webserver
API
Engine Module
Lexical Analyser Module
Syntactic Analyser Module
Style Analyser Module
1
1 - *
1
1 - *
1
1 - *
Language Token Definitions
Production Rule Definitions
Language Style Rules Definitions

THE ENGINE MODULE – SCAE-LIBRARY.CORE

The Engine Module – which correlates to the *scae-library.core* namespace, is the main driver of the library. It receives the request, and then calls all the necessary components to generate the response.

THE LEXICAL ANALYSER MODULE – SCAE-LIBRARY.TOKENISER

The step that occurs first is the Lexical Analyse of the source code to turn it into an ordered collection of tokens. The Engine passes to the *scae-library.tokeniser* namespace the source code and the token regex definitions from the rulebook. The tokeniser takes the regex definitions, combines them by ORing them together, and performs are regex search on the source code. It takes the matching sections of source code, and turns them into my token format, which is a hash-map.

- Each token has a key ":token-key" whose value is the key corresponding with the key associated with the matching regex.
- Each token has a key ":token-value" whose value is the string value of the token taken from the source code.
- Each token has a key ":token-type" whose value is ":token", denoting a regular token, or ":skip", denoting a token which may be discarded e.g. white spaces.

THE SYNTACTIC ANALYSER MODULE – SCAE-LIBRARY.ABSTRACT-SYNTAX-TREE

The Engine Module then passes the tokens to the Syntactic Analyser Module, the *scae-library.abstract-syntax-tree* namespace, along with the production rule definitions from the rulebook.

Firstly we take the *":scae-forward-declarations"* from the rulebook and evaluate them. This defines the production rule functions in the current namespace to that we can execute them.

Next we start parsing the production rules and tokenised code, starting with the production rule *":scae-entry-point"*.

As outlined in the "**Writing Rulebooks – Basic Format**" in the **User Manual**, each production rule is made up of a vector of vectors, with each inner vector representing an optional branch i.e. A|B.

An inner-vector/branch can be made up of either function calls to other production rules, keys matching the ":token-key" value of a token, or an empty vector representing an epsilon transition.

We examine each optional branch until we can get a branch that fully match the current sequence of tokenised code with that of the branch of the production rule.

If so we create a node, which is a hash-map.

- Each node has a key ":parsed-node-name" whose value is the string name of the matching production rule.
- Each node has a key ":parsed-node-result" whose value is a lazy-sequence of child nodes and tokens.

If not we move onto the next inner-vector/branch.

If no branches match then the production rule fails and we stop parsing and return nil.

THE STYLE ANALYSER MODULE – SCAE-LIBRARY.STYLE-ANALYSER

Next the Engine Module passes the generated Abstract Syntax Tree to the Style Analyser Module, the *scae-library.style-analyser* namespace, along with the :style-rules from the Style Rulebook.

Firstly we take the *":style-rules-function-definitions"* from the rulebook and evaluate them. This defines the style rule functions in the current namespace to that we can execute them.

Next we start parsing the Abstract Syntax Tree, starting from the root node, along with the *":node-based-style-rules"* from the Style Rulebook.

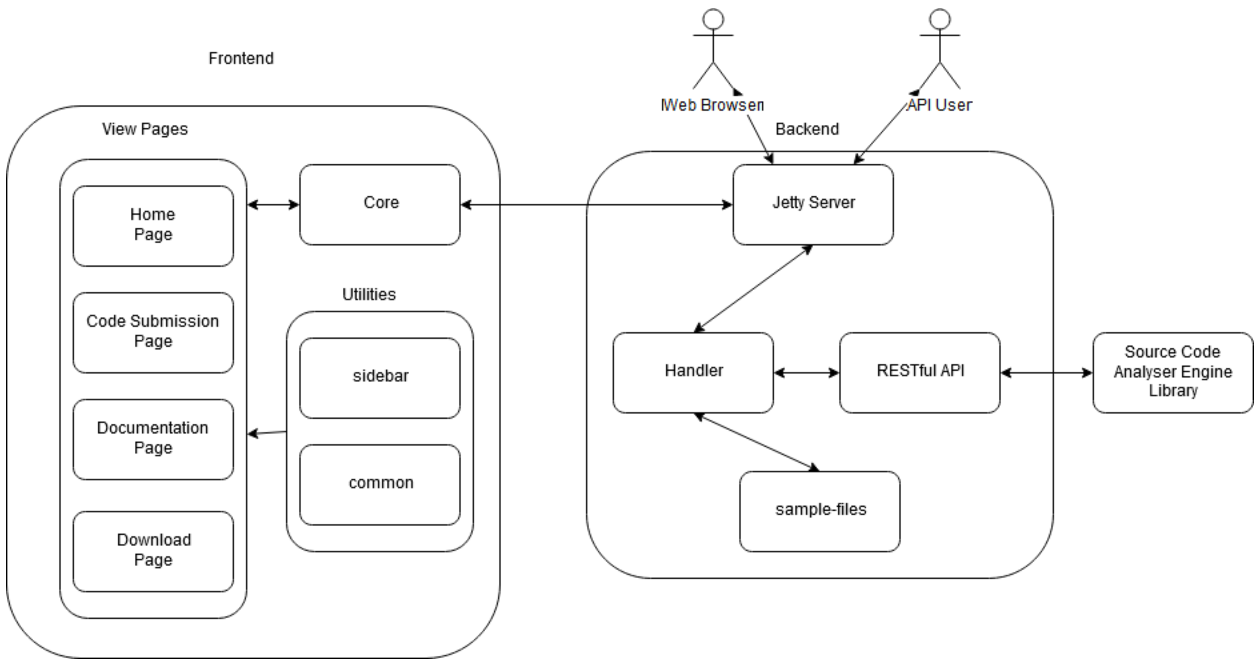
For the current node, we check if there are any Style Rules defined for it from the map *":node-based-style-rules"* in the Style Rulebook. If there are we apply those functions, which will examine the current node for any poor style, and if some are detected, return a string detailing the detected poor style, or an empty string if no poor style was detected.

As detailed in the "**Writing Rulebooks – Useful Style Rule Helper Functions**" in the **User Manual**, the *scae-library.node-ops* provides a variety of useful functions to make the process of navigating the Abstract Syntax Tree when writing Style Rules easier.

We then look at each child of the node and repeat this process for the entire tree, building up a collection of suggestions (or none) which will then be returned to the Engine Module, who in turn will return the result the calling process.

WEB APPLICATION: APPLICATION ARCHITECTURE

As previously stated, the Web Application www.source-code-analyser-engine.com, which is hosted on AWS, acts as both the home of the project and as an example of the kind of applications that some users can create to incorporate the Source Code Analyser Engine Library.



BACKEND

The backend contains all the server-side components of the Web Application. The backend is written in Clojure, a functional language built upon the JVM.

JETTY SERVER

The website is using a Jetty Server to serve content on port 80.

HANDLER

The handler takes requests that hits the Jetty Server and decides what to do with them.

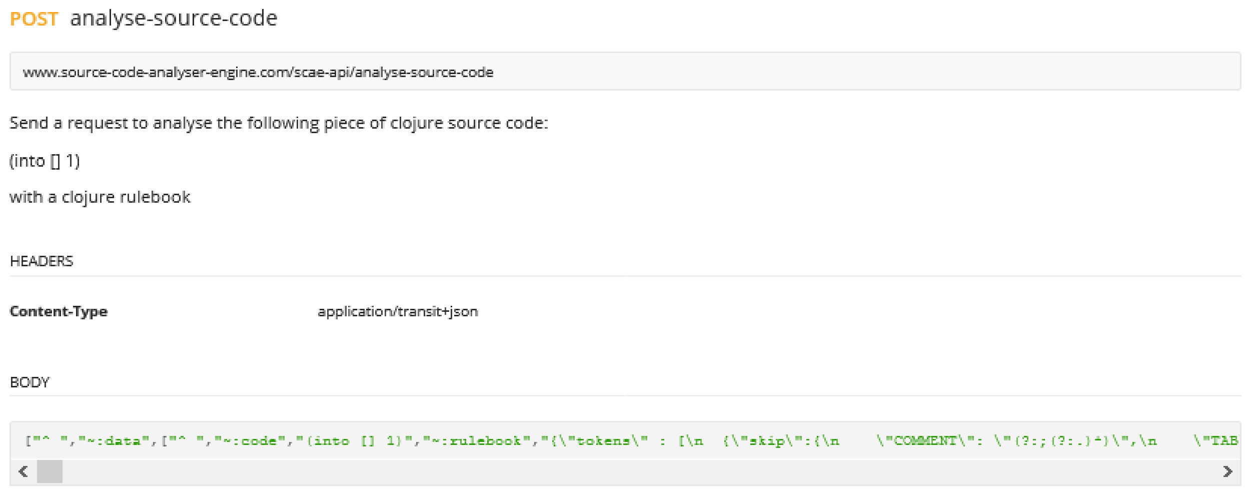
Some requests will be to navigate to a frontend view.

Some requests will be for the RESTful API.

Others will be to retrieve sample files.

RESTFUL API

The RESTful API has two functions, one to get Style Suggestions from the Library, the other to get the generated Abstract Syntax Tree from the Library. You can see the below documentation here.



POST analyse-source-code

www.source-code-analyser-engine.com/scae-api/analyse-source-code

Send a request to analyse the following piece of clojure source code:

(into [] 1)

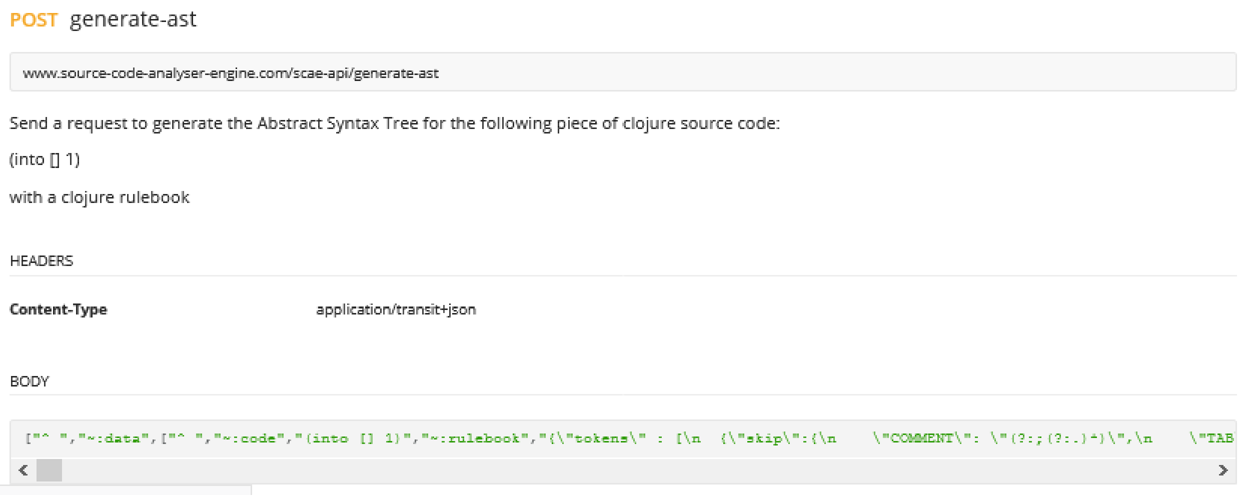
with a clojure rulebook

HEADERS

Content-Type	application/transit+json

BODY

```
["^ ","~:data",["^ ","~:code","(into [] 1)","~:rulebook","{\"tokens\" : [\n  {\"skip\":{\n    \"COMMENT\": \"(?:;(?:.)*)\",\n    \"TAB
```



POST generate-ast

www.source-code-analyser-engine.com/scae-api/generate-ast

Send a request to generate the Abstract Syntax Tree for the following piece of clojure source code:

(into [] 1)

with a clojure rulebook

HEADERS

Content-Type	application/transit+json

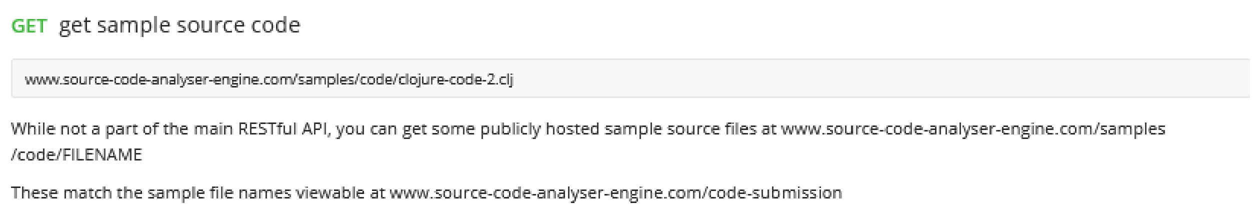
BODY

```
["^ ","~:data",["^ ","~:code","(into [] 1)","~:rulebook","{\"tokens\" : [\n  {\"skip\":{\n    \"COMMENT\": \"(?:;(?:.)*)\",\n    \"TAB
```

SAMPLE FILES

Receives a request for either code or rulebook sample files.

This will retrieve all stored sample files and return them to the caller.

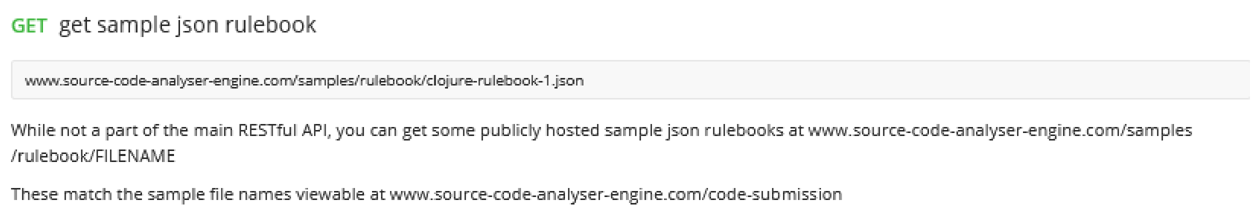


GET get sample source code

```
www.source-code-analyser-engine.com/samples/code/clojure-code-2.clj
```

While not a part of the main RESTful API, you can get some publicly hosted sample source files at www.source-code-analyser-engine.com/samples /code/FILENAME

These match the sample file names viewable at www.source-code-analyser-engine.com/code-submission



GET get sample json rulebook

```
www.source-code-analyser-engine.com/samples/rulebook/clojure-rulebook-1.json
```

While not a part of the main RESTful API, you can get some publicly hosted sample json rulebooks at www.source-code-analyser-engine.com/samples /rulebook/FILENAME

These match the sample file names viewable at www.source-code-analyser-engine.com/code-submission

FRONTEND

The frontend contains all the client-side components of the Web Application. The frontend is written in ClojureScript, a functional language that compiles down to JavaScript.

CORE

Defines the client-side routes and mounts the view pages.

VIEW PAGES

These are the pages that the User can see on the client.

HOME PAGE

This is the landing page of the Web Application. This acts as an introduction to the project and lists some of its features.

The page is located at www.source-code-analyser-engine.com

CODE SUBMISSION PAGE

This page allows the user to try the Library for themselves without any user-side set up required. They can either provide Source Code to be analysed and Style Rulebooks, or select from a range of pre-existing sample files, and then either Get Suggestions or the Abstract Syntax Tree for their data. More detailed information on this page is available in the **User Manual**.

The page is located at www.source-code-analyser-engine.com/code-submission

DOCUMENTATION PAGE

This page contains the documentation for the project.

The page is located at www.source-code-analyser-engine.com/documentation

DOWNLOAD PAGE

This page allows the user to download the latest version of both the Library and the Web Application.

Details on their use in located in the Installation Guide.

The page is located at www.source-code-analyser-engine.com/download

UTILITIES

These contain generic functionality that is useful in multiple parts of the frontend.

SIDEBAR

This contains the navigation sidebar that is displayed on all view pages.

COMMON

This is a namespace that contains some common helper functions, such as swapping in the value of an on change event into an atom.

CHALLENGES AND RESOLUTION

In this section I shall discuss some of the main challenges that I faced during the course of the project, and how I went about resolving them. To see more about my progress and the challenges that I faced on a Sprint (week-to-week) level, please refer to the Blog, where I go into greater detail on the challenges I faced while designing the system, especially from Blog Post 7 onwards.

THE TOKENISER

As outlined in Blog Post 7, one of the main challenges that I faced with the Tokeniser was to perform the user defined regex searches while preserving the ordering of the tokens. If I iteratively/ recursively performed the search for each token, then I would lose all ordering.

I solved the issue by extracting each user defined regex from the Style Rulebook, and combined them all together by ORing them. With this done, I could perform a single regex search for all tokens in the string at once, and get the separate token strings in order. With that ordered collection of strings, I could then convert them into my own token data structure as outlined in The Lexical Analyser Module – scae-library.tokeniser section.

ABSTRACT SYNTAX TREE

Generating the Abstract Syntax Tree presented a number of interesting challenges, as this would be the key data structure that the rest of my project would revolve around.

DEFINING THE FORMAT OF USER SUBMITTED PRODUCTION RULES

In order for the project to be modular, flexible and able to work for any programming language, I had to put in a lot of planning on how I would have the User design their production rules.

I needed the production rules to be both in a Clojure format, but also be capable of performing various Grammar Transitions.

As outlined in the The Syntactic Analyser Module – scae-library.abstract-syntax-tree section and in the **User Manual** I defined a single production rule as a vector. Within that vector would be a series of inner vectors, representing different branches OR'ed together. Each of these branches could then either contain calls to other production rules & tokens, or contain an empty vector to represent an Epsilon Transition.

PARSING THE PRODUCTION RULES

With the above format defined, I was able to design a system where I could recursively parse through the various inner branches of the production rule, and start matching them in order with the tokenised piece of source code.

EASE THE PROCESS OF NAVIGATING THROUGH THE ABSTRACT SYNTAX TREE

When I designed the The Style Analyser Module – scae-library.style-analyser, I noticed how awkward it could be for the user to define style rules a heavily nested data structure. So, I went ahead and designed the node-ops namespace, which defines a number of functions to make it both easier to navigate the Abstract Syntax Tree, and thus make it easier to create style rules.

DEPLOYMENT GUIDE

This section details the requirements and steps to install and run the system.

SYSTEM REQUIREMENTS

WEBSERVER

Recommended Specifications for the Server are:

Processor	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
Memory	1 GiB System Memory
Storage	5 GiB
Network	Physical Network Interface Card
OS	Ubuntu 16.04.4 LTS
Software	Java Version 1.8.0_161

CLIENT

Recommended Specifications for the Client are

Browser	HTML 5 Enabled Browser
Officially Supported Browsers	Firefox, Chrome

INSTALLATION GUIDE

LIBRARY

You have two options for retrieving the library jar file.

1. You can either download it from the website
2. Or you can build if from the source code
   a. Ensure you have Java 8 installed
   b. Ensure you have Leiningen installed
   c. Download the project from GitLab
   d. Navigate to the folder 2018-ca400-oconnb47/src/scae-library/
   e. Run the command: lein install
      i. On linux the jar will be placed in the .m2 directory in your home directory, where you can easily import it into other projects.
   f. Or you could run: lein jar
      i. This will place the jar into the /target folder of the current directory

Then refer to the documentation of whichever language/platform you are importing the library into for more details.

WEB APPLICATION

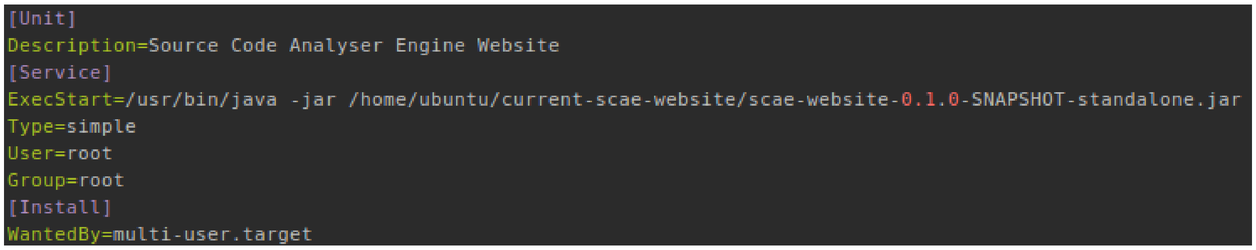
You have two options for retrieving the Web Application Uberjar.

1. You can either download it from the website
2. Or you can build if from the source code
   a. Ensure you have Java 8 installed
   b. Ensure you have Leiningen installed
   c. Ensure you have installed the library through the use of: lein install
   d. Ensure you have Yarn installed
   e. Download the project from GitLab
   f. Navigate to the folder 2018-ca400-oconnb47/src/scae-website/
   g. Run the command: yarn install --modules-folder resources/public/node_modules
   h. Run the command: lein uberjar
   i. The uberjar will be located in the target/ directory of the current directory

To run the webserver on Ubuntu run the command: sudo java -jar THE-WEBSITE-UBERJAR-NAME

To have the Web Application start up automatically on Ubuntu you can write a Service Script at

/lib/system/system/scae-website.service



```
[Unit]
Description=Source Code Analyser Engine Website
[Service]
ExecStart=/usr/bin/java -jar /home/ubuntu/current-scae-website/scae-website-0.1.0-SNAPSHOT-standalone.jar
Type=simple
User=root
Group=root
[Install]
WantedBy=multi-user.target
```

This allows you to start, stop, restart and check the status of the service by running

sudo service scae-website start

sudo service scae-website stop

sudo service scae-website restart

sudo service scae-website status

To enable auto starting of the service:

systemctl enable scae-website.service

TESTING

In this section I discuss the test plan that I developed early in the project to outline my testing strategy, and then the results of my testing.

TEST PLAN

PURPOSE

This purpose of this test plan is to describe the testing approach that will be used for the Source Code Analyser Engine project., with an emphasis on testing the core library of the project.

TEST OBJECTIVES

The objective of the tests is to verify the functionality of the Source Code Analyser Engine, with the main emphasis of the testing being directed towards the library portion of the project.

We will have a collection of automated tests that will be run before any code gets committed, and if a potential commit breaks a test the issue must be resolved before the code is committed.

TEST ASSUMPTIONS

- When creating the tests the tester will have access to expected input and output data that match closely to actual production data
- When any user testing is being done the user will have access to pre-written style rules and code samples to test them with
- There are Ubuntu Server virtual machines available as a staging environment that match the specs of the production AWS Ubuntu Server.

TEST PRINCIPLES

- The tests will emulate real world examples and data flow as much as possible
- Tests shall be carried out with invalid program input to test if the program can gracefully handle it and report back to the user any issues that arise
- Automated tests must pass before merging code to the codebase
- Testing will be repeatable, and where possible automated

TEST EXECUTION

The automated tests for the library shall be written using the Clojure Test framework. The tests can be executed using the command

lein eftest

These tests should be run and pass before any new code is merged.

For User & User acceptance testing, a survey should be provided to the user where they may give their feedback on the testing. This allows us to track any changes between rounds of user testing.

DEFECT TRACKING & REPORTING

If defects are detected before a commit, the defect should be fixed as part of the current task, the tests re-run, and if they pass then may the code be committed.

If defects are in the codebase (not due to the current task being worked on) then a new issue must be made on the GitLab issue board. The issue must describe the bug, and also give an indication to the priority that fixing the defect should take.

TEST ENVIRONMENT

The automated tests will run on the developer's laptop, using Leiningen (the Clojure build manager) to execute the tests.

For both stress-testing and full end-to-end testing of potential product breaking features shall be run on the staging VMs. This is both to minimize computing costs on the AWS servers, and to minimize the chance of the production webserver from going down.

"Typical" end-to-end / User testing may take place on the AWS production servers.

TESTING RESULTS

X

FUTURE PLANS

(symbol table – prototyping, testing viability, expand functionality)

REFERENCES

Leiningen: https://leiningen.org

Yarn: https://yarnpkg.com/en/