5/20/2018

Source Code Analyser Engine

User Manual

Bernard O'Connor
14367821
Dr David Sinclair

TABLE OF CONTENTS

OVERVIEW

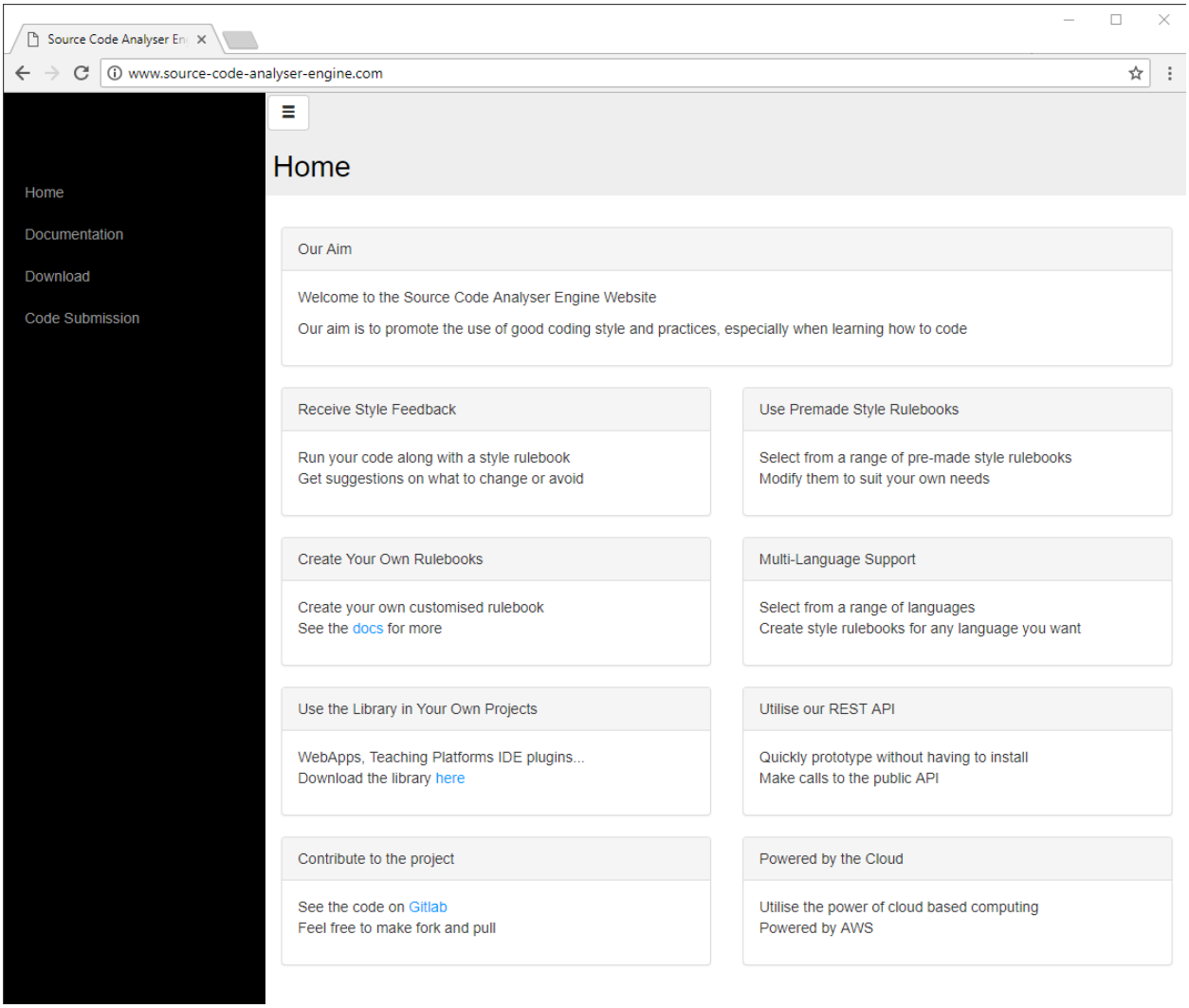
The aim of the project is to promote the use of good programming practices. The Source Code Analyser Engine achieves this by looking for poor programming style using what I call Node Based Styling.

The project is split into two, the Website and the Library, both written in Clojure(Script)

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.

THE WEBSITE

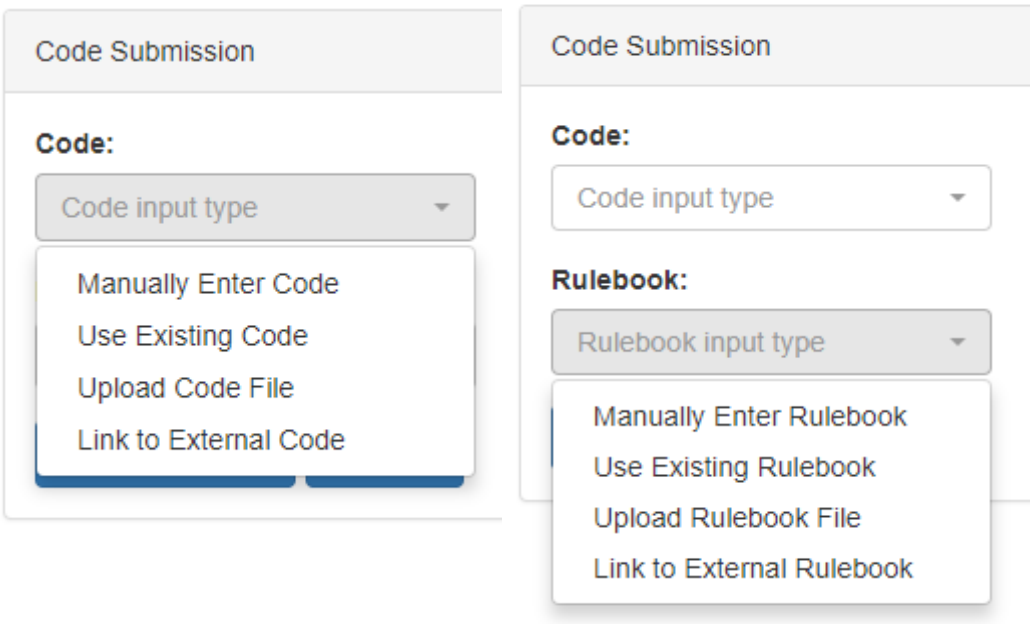


www.source-code-analyser-engine.com is the home of the project. From here you can access a variety of services such as additional documentation, downloading the project, and most importantly, trying out the project for yourself. The website is currently deployed on AWS at the above URL.

CODE SUBMISSION PAGE

Here you have the opportunity to try out the project for yourself. You can either see how the project works by using sample pieces of source code and style rulebooks, or you can use your own code.

TYPES OF INPUT



You can choose from a variety of methods for how you want to input your data.

"Manually Enter Code/Rulebook" lets you directly enter in some code or a rulebook.

"Use Existing Code/Rulebook" brings up a dropdown where you can select a sample piece of code.

"Upload Existing Code/Rulebook" allows you to upload a source file or a JSON Rulebook.

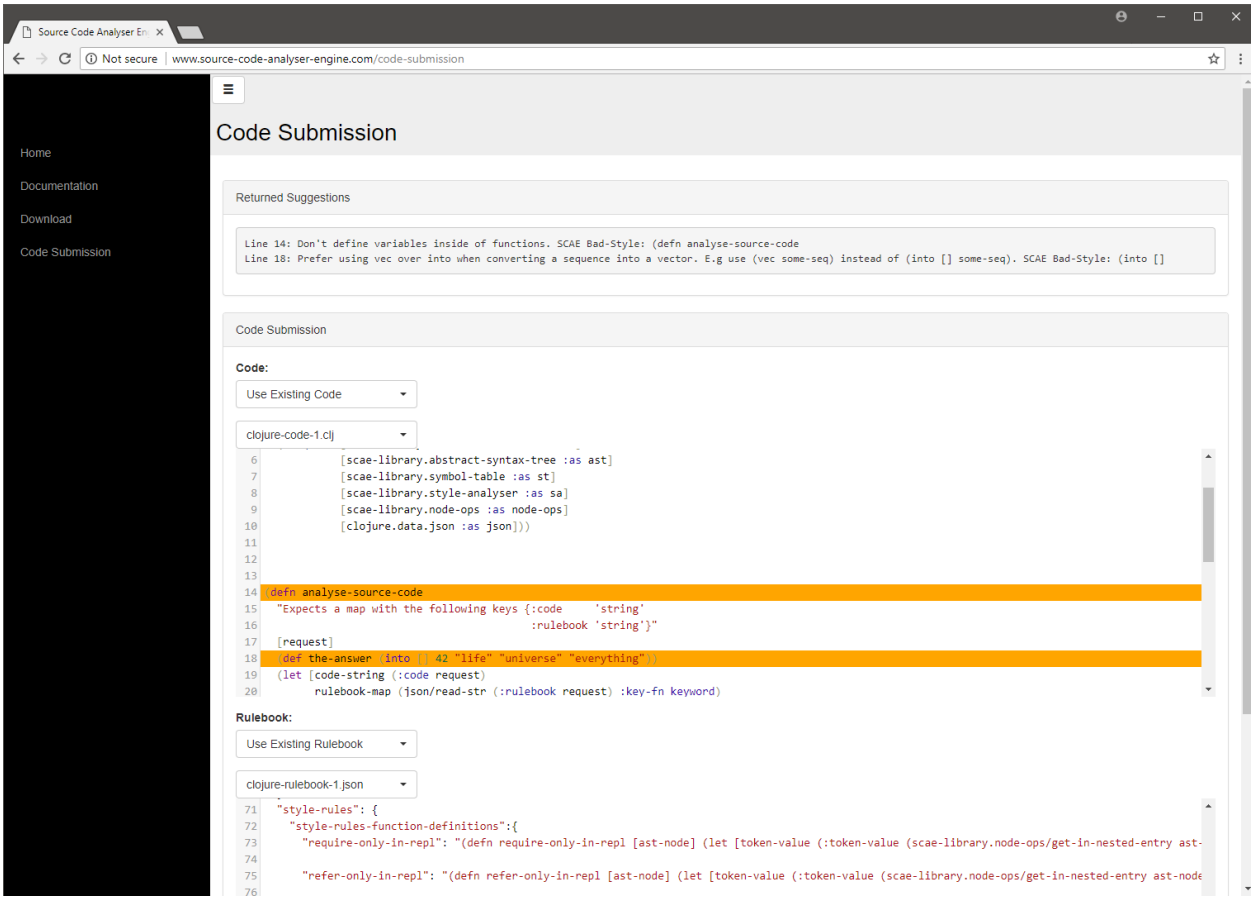
"Link to External Code/Rulebook" allows you to enter a URL to either a source file or a JSON Rulebook.

A text editor will be displayed for each option showing you the current value of the code/rulebook you have selected. Note that for the existing URL option the editor is only updated when you submit your code.

GET STYLE SUGGESTIONS

When you have entered in your code and rulebook you can click on the "Get Suggestions" button. This takes the data you've just entered and passes it to the Source Code Analyser Engine to checked for poor style.

As you can see below, we have detected some examples of poor style. Suggestion are returned outlining the source of the poor style, along with highlighting where the poor style has taken place.



It is important to note here for any users wishing to incorporate the Library into their own projects or write their own rulebooks, that the detection of Line Numbers and highlighting is all done on this website's frontend.

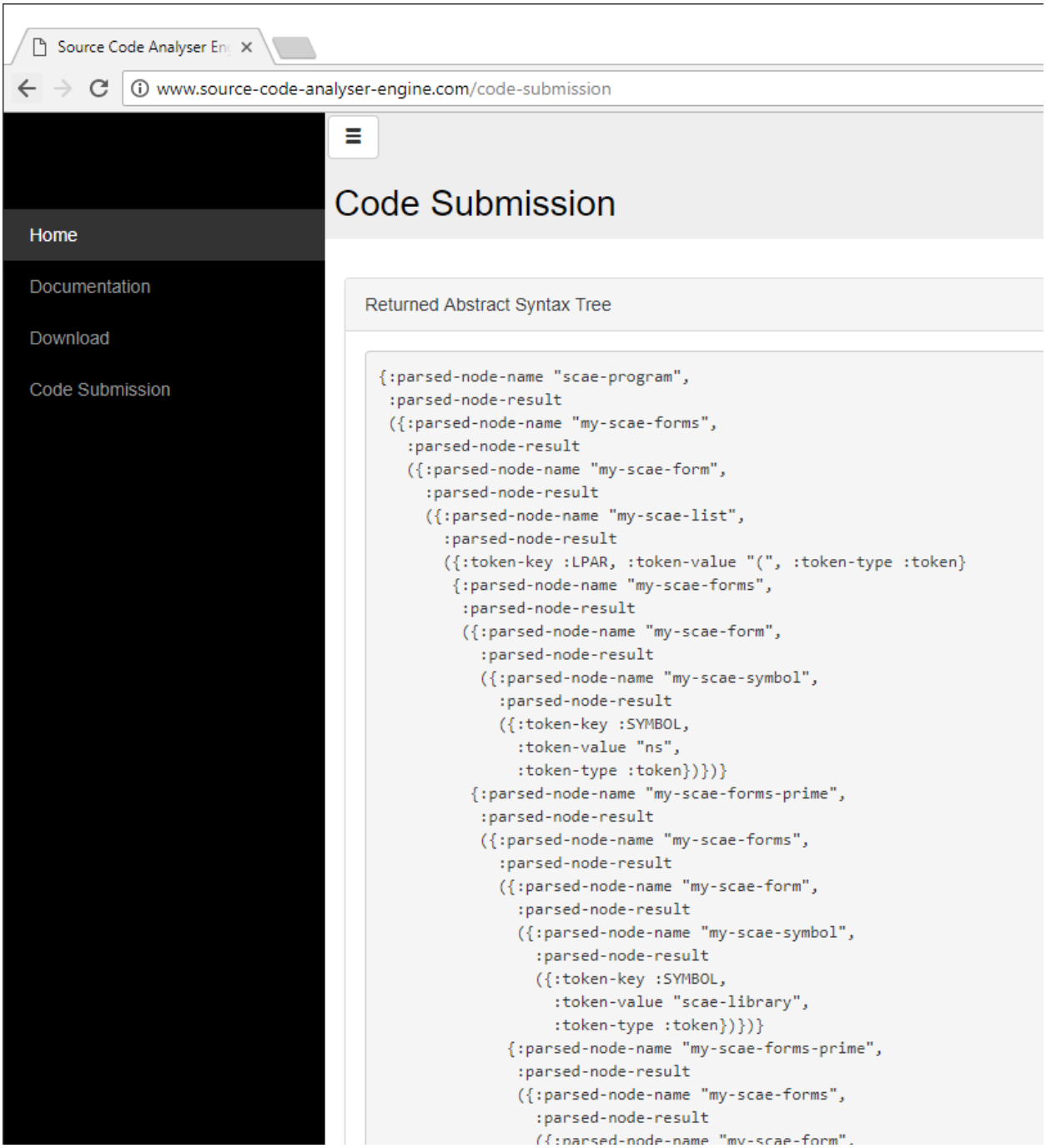
The library loses all concept of line numbers during the tokenisation process. I get around this on the website by inserting strings that I can search for, number and highlight at the end of the returned suggestions, e.g. "SCAE Bad-Style: (into []".

It is not mandatory to include this, and if it is not included then the website will work fine without suggestion line numbering and highlighting. Just note that either all style suggestions in a rulebook should have this kind of string at the end, or else none of them should. A mixture of both will result in the mis-numbering and highlighting of suggestions.

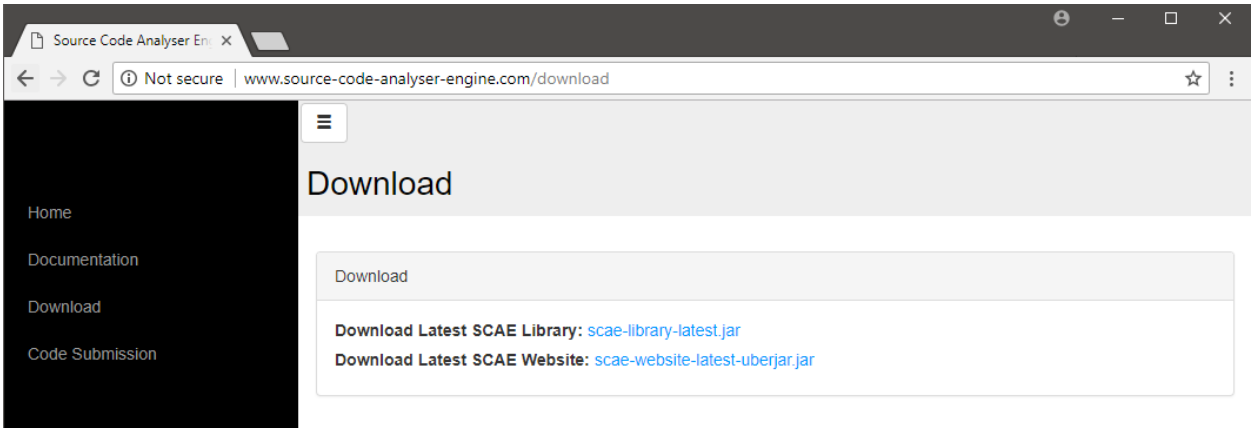
GET THE ABSTRACT SYNTAX TREE

If you would like, you can retrieve the Abstract Syntax Tree that is generated for your piece of code. It is from this Abstract Syntax Tree that we apply the Node Based Styling Rules to detect any poorly styled code.

Getting the Abstract Syntax Tree is especially useful if you are developing your own Style Rules, which is more in depth later.



DOWNLOAD PAGE



You can find the latest version of both the Library and the Website available for download.

www.source-code-analyser-engine.com/download

The library jar can be imported into any standalone project you have.

The website uber-jar is the full version of the Source Code Analyser Website. It requires sudo permission for the server to be allowed to run on port 80.

```
sudo java -jar WEBSITE-UBERJAR
```

You can then view the running website by opening your web-browser and going to localhost or 127.0.0.1

DOCUMENTATION PAGE

Here you can view both this User Manual and the Technical Guide, along with a link to the RESTful API documentation

www.source-code-analyser-engine.com/documentation

documenter.getpostman.com/view/4021417/RW87op7E

RESTFUL API

The full documentation for the RESTful API with example requests and responses can be found here.

This documentation can also be found from the documentation page of the website.

ANALYSE SOURCE CODE

To make an API request to analyse the source code simply make a POST request to www.source-code-analyser-engine.com/scae-api/analyse-source-code with a map as a parameter in the format:

```
{:code "your code as a string"
 :rulebook "a json rulebook here "}
```

The response will be a collection of strings, with each string denoting a detected piece of poor style

GENERATE ABSTRACT SYNTAX TREE

To make an API request to generate the Abstract Syntax Tree for the source code simply make a POST request to www.source-code-analyser-engine.com/scae-api/generate-ast with a map as a parameter in the format:

```
{:code "your code as a string"
 :rulebook "a json rulebook here "}
```

The response will be a map denoting the Abstract Syntax Tree

```
{:parsed-node-name "scae-program",
 :parsed-node-result
 ({:parsed-node-name "my-scae-forms",
   :parsed-node-result
   ({:parsed-node-name "my-scae-form",
     :parsed-node-result
     ({:parsed-node-name "my-scae-literal",
       :parsed-node-result
       ({:token-key :STR,
         :token-value "\"Hello World\"",
         :token-type :token})})}
    {:parsed-node-name "my-scae-forms-prime",
     :parsed-node-result
     ({:parsed-node-name "my-scae-forms", :parsed-node-result []})})})}
```

THE LIBRARY

The library has two major functions that you will be interacting with, analyse-source-code and generate-ast in the core namespace of the library. These are the two functions that end up being called from the API that we've mentioned before. They both take in a single parameter of a map, with the keys :code and :rulebook. And as outlined previously, they return either a collection of strings for the suggestions, or else a hash-map containing the Abstract Syntax Tree.

WRITING RULEBOOKS

BASIC FORMAT

As mentioned, the Rulebooks are taken in as a JSON file, and are made up of the following format.

```
{"tokens" : [
  {"skip":{
    "COMMENT": "(?:;(?:.)*)",
    "TAB" : "\t",
    "NEWLINE" : "\n",
    "CARRIAGE_RETURN" : "\r",
    "FORM_FEED" : "\f"
  }},


  {"token" : {
    "LPAR" : "\\(",
    "RPAR" : "\\)",
    "LSQU" : "\\[",
    "RSQU" : "\\]",
    "LCUR" : "\\{",
    "RCUR" : "\\}"}}

],
  "productions": {
    "scae-forward-declarations":"(declare scae-program YOUR-production-rule-1 YOUR-production-rule-2)",
    "scae-program": "(defn scae-program [] [['(YOUR-production-rule-1)] [[]] ])",
    "scae-entry-point": "(scae-program)",

    "YOUR-production-rule-1": "(defn YOUR-production-rule-1 [] [['(YOUR-production-rule-2) :TOKEN-KEY-4)] [[]] ])",
    "YOUR-production-rule-2":"(defn YOUR-production-rule-2 [] [[:TOKEN-KEY-1 :TOKEN-KEY-2] [:TOKEN-KEY-1 :TOKEN-KEY-3] ])"


  },
  "style-rules": {
    "style-rules-function-definitions":{
      "YOUR-style-function-1" : "(defn YOUR-style-function-1 [ast-node] your logic here. Return a string with a suggestion or an empty string)"
    },
    "node-based-style-rules" : {
      "YOUR-node-name-to-apply-rules-to-1" : "[YOUR-style-function-1]"
      }
  }}
```

Note this is a stripped-down version of a rulebook for explanation purposes. See the code submission page for full functional rulebook examples.

1. The first entry we have is the "tokens" entry. This contains a list of maps. In this list we can have multiple "skip" or "token" maps. The keys of these maps will be used as the name of the corresponding token. The values are strings denoting Java Patterns that will be used to perform a regex search to generate the value of the token. Skip tokens will ultimately be filtered out and ignored from the Abstract Syntax Tree
2. The next entry is the "productions" map entry. These define the production rules that will be used to generate the Abstract Syntax Tree.
    a. The first entry is a special entry called "scae-forward-declarations". Its value is the "declare" function containing the name of every production rule that will be defined. This allows the easy calling of functions that are defined at a later stage.

b. The next two entries are "scae-program" and "scae-entry-point". "scae-entry-point" is a special entry whose value contains a function call to the initial production rule of the Grammar, which in this case we have called it "scae-program".
c. The rest of the entries in "productions" are the production rules of the grammar. They take the format of a key (which is recommended to make the name of the function for readability sake) and a string value which is a Clojure function defining the production rule.
    i. In the Source Code Analyser Engine production rules are defined as a vector of vectors.
        1. Each inner vector is an optional branch that we can go down when parsing. So, having two inner vectors is the equivalent of saying Branch-A OR Branch-B. If a branch does not match the stream of tokens, we reject that branch and move on to check the next branch.
        2. Within an inner-vector / branch, we will have either a mixture of function calls to another production rules and some token-keys, or an empty vector. An empty vector represents an Epsilon Transition. Note that when we make a function call within the vector that we must "quote" the function call with an apostrophe, e.g. ['(func-call) :RANDOM-TOKEN]
d. See the section of Useful Production Rule Design Patterns for more information on designing production rules

3. The last section is the "style-rules" map. This map contains firstly the "style-rules-function-definitions" map and then the "node-based-style-rules" map.
    a. The "style-rules-function-definitions" is a map containing the functions that you will use to detect bad style within a node of an AST. The functions will take one parameter, the *"ast-node"* you are examining. After you perform your checks on the node, return either a string detailing suggestion to improve the code, or return an empty string. See the section on Useful Style Rule Helper Functions for details on the "node-ops" namespace that help make dealing with the generated Abstract Syntax Tree easier
    b. The "node-based-style-rules" map has a key which will match the node name of the production rule that you wish to apply Node Based styling to. The value will be a vector listing the name of the style functions (defined previously and mentioned below) that you wish to apply to the Node

USEFUL PRODUCTION RULE DESIGN PATTERNS

To represent an epsilon transition *(ε)* we have the only entry to an inner vector be another empty vector. So the inner vector would be *[[]]*

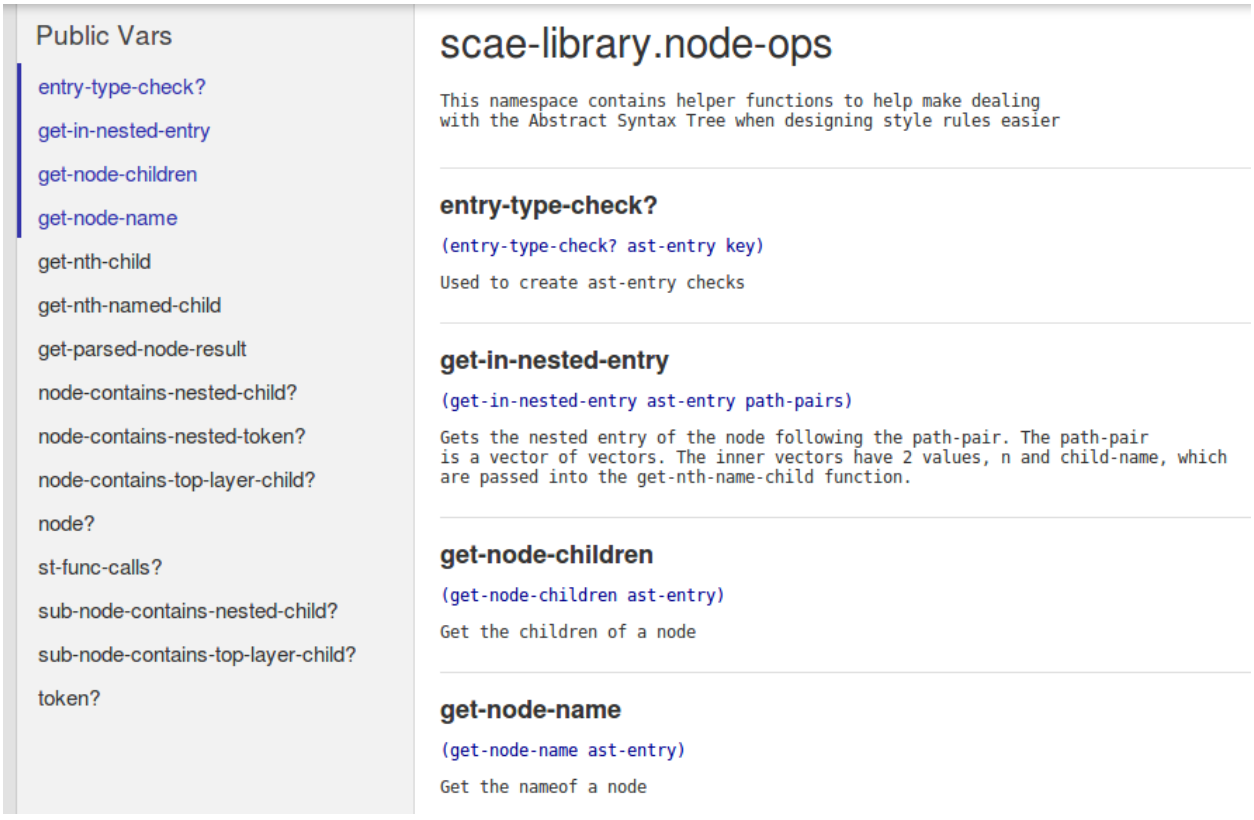
To represent the transition zero or more transition *(my-scae-form)** we can use the following production design

```
"my-scae-forms": "(defn my-scae-forms [] [['(my-scae-form) '(my-scae-forms-prime)] [[]] ])",
"my-scae-forms-prime":"(defn my-scae-forms-prime [] [['(my-scae-forms)] [[]] ])",
```

To represent one or more *(my-scae-form)+* in the above example we would remove the epsilon transition from the "my-scae-form" production rule

To represent an OR *(A|B)* operation in a single branch it is recommended to place that OR operation into its own production rule so that the first inner vector contains A and the second inner vector contains B

USEFUL STYLE RULE HELPER FUNCTIONS



Public Vars

- entry-type-check?
- get-in-nested-entry
- get-node-children
- get-node-name
- get-nth-child
- get-nth-named-child
- get-parsed-node-result
- node-contains-nested-child?
- node-contains-nested-token?
- node-contains-top-layer-child?
- node?
- st-func-calls?
- sub-node-contains-nested-child?
- sub-node-contains-top-layer-child?
- token?

get-nth-child

```
(get-nth-child ast-node n)
```

Get the nth child of the ast node. Uses zero indexing (0 is first position). Ignores symbol table function calls and line-numbers

get-nth-named-child

```
(get-nth-named-child ast-node n child-name)
```

Retrieve the nth instance of the target child. Uses zero indexing

get-parsed-node-result

```
(get-parsed-node-result ast-node)
```

This returns the parsed node result. If the parsed node result is a string, it reads the string and returns the eval'd result. This is due to elsewhere in the program, the style analyser, having to quote the parsed node result so that internal hash-maps would not try to be evaluated as a function if they were at the head of a list and eval were called on them

node-contains-nested-child?

```
(node-contains-nested-child? ast-node child-name)
```

Check if a node contains a certain child. Note that child-name is the name of the node / token and not its value.

node-contains-nested-token?

```
(node-contains-nested-token? ast-entry token-key token-value)
```

Returns true if the node contains a token that has a value of token-value with key of token-key anywhere in its children or their sub-trees

Public Vars

- entry-type-check?
- get-in-nested-entry
- get-node-children
- get-node-name
- get-nth-child
- get-nth-named-child
- get-parsed-node-result
- node-contains-nested-child?
- node-contains-nested-token?
- node-contains-top-layer-child?
- node?
- st-func-calls?
- sub-node-contains-nested-child?
- sub-node-contains-top-layer-child?
- token?

node-contains-top-layer-child?

```
(node-contains-top-layer-child? ast-node child-name)
```

Check if a node contains a certain direct child. Note that child-name is the name of the node / token and not its value.

node?

```
(node? ast-entry)
```

Check if the current ast-entry is of type node

st-func-calls?

```
(st-func-calls? ast-entry)
```

Check if the current ast-entry is of type st-func-calls

sub-node-contains-nested-child?

```
(sub-node-contains-nested-child? ast-node sub-node-name child-name)
```

Check if a sub node contains a certain child. Note that child-name is the name of the node / token and not its value.

sub-node-contains-top-layer-child?

```
(sub-node-contains-top-layer-child? ast-node sub-node-name child-name)
```

Check if a sub node contains a certain direct child. Note that child-name is the name of the node / token and not its value.

node?

st-func-calls?

sub-node-contains-nested-child?

sub-node-contains-top-layer-child?

token?

token?

`(token? ast-entry)`

Check if the current ast-entry is of type token

ABSTRACT SYNTAX TREE FORMAT

The Abstract Syntax Tree is a nested hash-map of nodes as described by the production rules.

Each node has a ":parsed-node-name" which has a string value of the name of the node.

Each node has a ":parsed-node-result" which contains a list of nodes and tokens which are children of the current node.

Each token has a ":token-key" which is the key representing the name of the token, matching the key of the regex that generated the token.

Each token has a ":token-value" which is the string representation of the token from the source code.

Each token has a ":token-type" which declares the type of the token, either ":token" or ":skip". Skip tokens are filtered out and not placed into the AST , e.g. whitespaces and newline characters.

```
{:parsed-node-name "scae-program",
 :parsed-node-result
 ({:parsed-node-name "my-scae-forms",
   :parsed-node-result
   ({:parsed-node-name "my-scae-form",
     :parsed-node-result
     ({:parsed-node-name "my-scae-literal",
       :parsed-node-result
       ({:token-key :STR,
         :token-value "\"Hello World!\"",
         :token-type :token})})}
    {:parsed-node-name "my-scae-forms-prime",
     :parsed-node-result
     ({:parsed-node-name "my-scae-forms", :parsed-node-result []})})})}
```

REFERENCES

- Details on java regex patterns:
    - https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html