

# Bachelor Thesis

Catherine Mathieu

Development of animated visualizations of code execution  
using abstract syntax tree transformations  
and web technologies

Catherine Mathieu

Development of animated visualizations of code  
execution using abstract syntax tree transformations  
and web technologies

Bachelor Thesis based on the examination and study regulations  
for the Bachelor of Engineering degree programme  
*Bachelor of Science Information Engineering*  
at the Department of Information and Electrical Engineering  
of the Faculty of Engineering and Computer Science  
of the University of Applied Sciences Hamburg  
Supervising examiner: Prof. Dr. Klaus Jünemann  
Second examiner: Prof. Dr.-Ing. Karin Landefeld

Day of delivery: July 1st 2020

**Catherine Mathieu**

**Title of Thesis**

Development of animated visualizations of code execution using abstract syntax tree transformations and web technologies

**Keywords**

Abstract Syntax Tree, Algorithm, Code Transformation, Debugging, E-learning, JavaScript, Memory Management, Web

**Abstract**

The work presents a prototype of an interactive e-learning platform for exploring programming and algorithms. The application aims to support the user to understand a code by illustrating its variables and creating graphical visualizations from input code. The approach explored in this work is based on code transformation and generation using abstract syntax tree techniques.

**Catherine Mathieu**

**Titel der Abschlussarbeit**

Entwicklung animierter Visualisierungen der Codeausführung mit Hilfe von abstrakten Syntaxbaumtransformationen und Webtechnologien

**Stichworte**

Abstrakter Syntaxbaum, Algorithmus, Quelltexttransformation, Debugging, E-Learning, JavaScript, Memory Management, Web

**Abstrakt**

Die Arbeit stellt einen Prototyp einer interaktiven E-Learning-Plattform zur Erkundung von Programmierung und Algorithmen vor. Die Anwendung zielt darauf ab, den Benutzer dabei zu unterstützen, einen Code zu verstehen, indem er seine Variablen anschaulich identifiziert und grafische Visualisierungen aus dem Eingabecode erstellt. Bei dem in dieser Arbeit behandelten Ansatz handelt es sich um eine Quelltexttransformation und -generierung, die abstrakte Syntaxbaumverfahren verwendet.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Codes</b>	<b>x</b>
<b>Acronyms</b>	<b>xi</b>
<b>Glossary</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Intention . . . . .	1
1.2 Vision . . . . .	2
1.3 Content . . . . .	2
<b>2 Requirements</b>	<b>3</b>
2.1 Product requirements . . . . .	3
2.1.1 Accessibility . . . . .	4
2.1.2 User experience . . . . .	4
2.1.3 Target language . . . . .	4
2.2 Architectural requirements . . . . .	5
2.2.1 Extensibility . . . . .	5
2.2.2 Testability . . . . .	6
2.3 Functional requirements . . . . .	6
2.3.1 Common programming concepts . . . . .	6
2.3.2 Set of basic code examples . . . . .	7
2.3.3 Code edition . . . . .	8
2.3.4 Variable visualization . . . . .	8
2.3.5 Custom graphical visualization . . . . .	9
2.3.6 Code execution navigation . . . . .	10

2.3.7	User interface feedback . . . . .	10
2.4	User interface . . . . .	11
2.4.1	Application bar . . . . .	12
2.4.2	Code library . . . . .	12
2.4.3	Playback . . . . .	13
2.4.4	Code editor . . . . .	14
2.4.5	Variable visualization . . . . .	15
2.4.6	Graphical visualization . . . . .	16
2.5	Approaches for code execution visualization . . . . .	17
2.5.1	Scripted animation . . . . .	18
2.5.2	Tracing instruction . . . . .	18
2.5.3	Debugger supervision . . . . .	19
2.5.4	Code transformation . . . . .	20
<b>3</b>	<b>Theory</b>	<b>22</b>
3.1	Abstract syntax tree (AST) . . . . .	22
3.1.1	Code parsing . . . . .	23
3.1.2	Anatomy of an AST . . . . .	23
3.1.3	AST traversal . . . . .	24
3.1.4	AST transformation and generation . . . . .	25
3.2	Advanced programming concepts . . . . .	25
3.2.1	Memory management . . . . .	25
3.2.2	Variable scope . . . . .	26
3.2.3	Variable hiding . . . . .	26
3.2.4	Closure . . . . .	26
<b>4</b>	<b>Design</b>	<b>28</b>
4.1	Technologies . . . . .	30
4.1.1	Web platform . . . . .	30
4.1.2	User interface library . . . . .	31
4.1.3	User interface style library . . . . .	33
4.1.4	Open source . . . . .	34
4.1.5	Text editor library . . . . .	34
4.1.6	Test libraries . . . . .	34
4.1.7	Module bundler and package manager . . . . .	35

4.2	Target language . . . . .	36
4.2.1	Language comparison . . . . .	36
4.2.2	JavaScript . . . . .	39
4.2.3	Babel . . . . .	39
4.2.4	AST Explorer . . . . .	40
4.3	Architecture . . . . .	41
4.3.1	Modules . . . . .	41
4.3.2	Process data flow . . . . .	44
<b>5</b>	<b>Implementation</b>	<b>48</b>
5.1	Parse code . . . . .	48
5.2	Augment AST . . . . .	50
5.3	Generate augmented code . . . . .	53
5.4	Generate execution steps . . . . .	53
<b>6</b>	<b>Test</b>	<b>55</b>
6.1	Unit tests . . . . .	55
6.2	Integration tests . . . . .	56
6.3	End-to-End tests . . . . .	59
6.4	Code testability . . . . .	61
<b>7</b>	<b>Conclusion</b>	<b>62</b>
7.1	Accomplishments . . . . .	62
7.2	Future works . . . . .	63
	<b>Bibliography</b>	<b>64</b>
<b>A</b>	<b>Appendix</b>	<b>69</b>
<b>B</b>	<b>Appendix</b>	<b>70</b>
<b>C</b>	<b>Appendix</b>	<b>74</b>
<b>D</b>	<b>Appendix</b>	<b>76</b>
	<b>Declaration</b>	<b>77</b>

# List of Figures

2.1	User interface layout overview. . . . .	11
2.2	Mockup of the application bar section. . . . .	12
2.3	Mockup of the code library section. . . . .	13
2.4	Mockup of three different states of the playback section. . . . .	13
2.5	Mockup of the code editor section. . . . .	14
2.6	Mockup of the variables visualization section. . . . .	15
2.7	Mockup of the graphical visualization section. . . . .	16
2.8	User interface overview. . . . .	17
2.9	Screenshot of an example in Python Tutor. . . . .	20
3.1	AST of an array declaration constructed by the JavaScript parser Babylon7 used by Babel. . . . .	24
4.1	Use case diagram of a user examining a code fragment execution. . . . .	29
4.2	Screenshot of an abstract syntax tree transformation example taken from AST Explorer. . . . .	41
4.3	Component diagram from a high level structure perspective. . . . .	42
4.4	Activity diagram of the operational flow for building a visualization. . . . .	45
4.5	Data representation at each step of the process flow. . . . .	46
5.1	Example of an abstract syntax tree created with the parser Babylon7. . . . .	49
5.2	Structure of a simple Abstract Syntax Tree (AST) generated with Babel. . . . .	50
5.3	Structure of a simple augmented AST generated by Babel. . . . .	52
5.4	Example of an execution steps array. . . . .	54
6.1	Result of multiple unit tests for clamp function. . . . .	56
6.2	Summary of all Jest tests. . . . .	56
6.3	Result of tests for a variable declaration. . . . .	57
6.4	Result of tests for all code fragments. . . . .	59

6.5	Result of puppeteer tests for the playback controls. . . . .	60
6.6	A verbose explanation of a failed test from Puppeteer. . . . .	61
A.1	Example of a cluttered code taken from Algorithm-visualizer[56]. . . . .	69
C.1	Advanced memory visualization of a function call that will generate a closure. . . . .	74
C.2	Advanced memory visualization of a closure. . . . .	74
C.3	Advanced memory visualization of an object with values and references. . . . .	75



# List of Tables

2.1	Comparison of approaches to visualize code execution. . . . .	17
4.1	Comparison of popular web frameworks relative to custom criteria. . . . .	31
4.2	Comparison of popular programming languages relative to custom criteria. . . . .	36
B.1	Variable visualization for block scopes. . . . .	70
B.2	Variable visualization for object. . . . .	71
B.3	Variable visualization for recursion. . . . .	71
B.4	Variable visualization for variable hiding. . . . .	72
B.5	Variable visualization for closure. . . . .	72
B.6	Variable visualization for graph. . . . .	73

# List of Codes

3.1	Simple array declaration. . . . .	23
5.1	Example of a simple code fragment. . . . .	48
5.2	Import and function call of babel/parser. . . . .	49
5.3	Import and function call of babel/traverse. . . . .	51
5.4	Code transformation of a variable declaration. . . . .	51
5.5	Import and function call of babel/generator. . . . .	53
5.6	Example of an augmented code fragment. . . . .	53
5.7	Code handling of a variable declaration. . . . .	53
6.1	Code example of a unit test. . . . .	56
6.2	Code example of an integration test. . . . .	57
6.3	Integration test code for validating all code fragment outputs. . . . .	57
6.4	Code example of a E2E test. . . . .	60
D.1	Example of a code to be used in AST Explorer website using babelv7 parser. . . . .	76

# Acronyms

API	Application Programming Interface.
AST	Abstract Syntax Tree.
CSS	Cascading Style Sheets.
CST	Concrete Syntax Tree.
DOM	Document Object Model.
E2E	End-to-End.
ES	ECMAScript.
HTML	HyperText Markup Language.
IDE	Integrated Development Environment.
JSX	JavaScript Extensible Markup Language.
KB	Kilobyte.
MVC	Model-View-Controller.
NPM	Node Package Manager.
OS	Operating Systems.
UI	User Interface.

# Glossary

augmented code	The code transformed by the code generation process.
execution steps	List of variables states and meta-programming information at each point of the execution of a code.
modules	Parts of a JavaScript program that can be imported or exported when requested.
target language	The coding language used by the user and to be analyzed by the application.
tracing instructions	Functions that are added to a code in order to display a visualization.

# 1 Introduction

*"Tell me and I forget, teach me and I may remember, involve me and I learn."*

-Benjamin Franklin

## 1.1 Intention

Learning to code is a great challenge for novice software developers and engineers. It is an enterprise that requires a lot of practice, precious time and efforts. The main intention of this work is to provide a new tool to help ease the steep learning curve of coding by complementing the currently available resources.

To learn a coding language, various resources are available. Books are good because they are usually detailed and contain a lot of explanations. They often have the advantage of using pseudo-code to smoothly explain the programming concepts, or even be available in a lots of specific programming languages. Yet, not everyone can learn with only books. Another traditional way is to attend a lecture that usually comes with both theory and practical component. Some find more useful to follow online tutorials at their own pace. There exists some great learning platforms like Khan Academy[24], where thousand of online videos are available. More specific to algorithms and data structure, there exist multiple websites and mobile applications that teach with visual representations. Overall, everyone has its own way to learn and it often involves a mix of multiple mediums.

Despite all the available resources, novices in software development will eventually face a code that they don't understand. Some will try to simplify the code by separating it into smaller pieces. Others might use the console to print the variables or just use a debugger and go through the code. Although, using a debugger sometime requires non-trivial knowledge about the functionalities of an [Integrated Development Environment \(IDE\)](#).

Would it be enlightening to have someone or something that would guide and explain the code? The helper could highlight the code step by step as the execution process at desired speed. The value of each variable could be displayed aside from the code and dynamically updated. As a final touch, a visual animation of what happens in the code could be displayed. And what if this helper was a software platform that could analyze any code you throw at it and teach you interactively how it is executed?

### 1.2 Vision

The aim of this work is to create a simple and intuitive prototype tool that would help people to learn basic programming concepts and understand various algorithms through an interactive platform. The tool should present a collection of popular code fragments and allow code edition and copy-pasting. It seems that this idea wasn't yet implemented and the goal of this work is to create a prototype to evaluate how realizable it is to build such a tool.

The mantra of this project is synergy between two important concepts: explanatory power and simplicity. The power to explain can often bring complexity, while simplicity is needed to avoid confusion and avoid learning barrier. There is a danger awaiting any e-learning platform like this one, which is to incorrectly assume what the learner already knows or understand. It is important to get into the skin of a beginner (rather than to get under their skin) and consider that they might know very little about programming.

### 1.3 Content

After the introduction, a requirements section describes the specifications for creating this prototype. Then, a theory section clarifies some advanced concepts to better understand the following design and implementation sections. Next is a section describing how tests support the prototype. Finally, the conclusion summarizes the accomplishments and proposes an opening to further works.

## 2 Requirements

The main objective of the expected application is to guide the user through the step by step execution of a code fragment. The application should divide the code execution into steps and display the variables states for each of them. This guidance should include dynamic highlight of the code fragment and visual animation of the variables values to help the user understand the execution.

In this section, requirements are presented in a general to specific order. They are separated into three categories: product, architecture and functionality. A level of importance is assigned to each requirement to indicate their priority. This level considers the effort required to work on a requirement and the added value for the application relative to the work of this thesis. A reasonable scope for a minimal viable solution would include all high priority requirements. Any medium requirement completion would be a bonus. Low requirements are out of scope, but would be valuable in the long term development of the application.

A later segment of this section illustrates how each part of the [User Interface \(UI\)](#) should look like and describes them in detail.

Finally, one of the challenges of creating such an application is the analysis of code. The last section presents various approaches to achieve visualization of code execution and explains the selected one.

### 2.1 Product requirements

The product requirements describe what the user should experience, how the application should be used and what are its general characteristics.

### 2.1.1 Accessibility

The targeted users of this application are students and novice in software development field. If the application would need multiple steps of installations, it could discourage many users to use it. Hence, the access to the application should be simple and direct.

ID	Requirement	Priority
1.1	Be easy to access or install	High
1.2	Support Microsoft Windows, macOS and Linux platforms	High

### 2.1.2 User experience

The UI should be self-explanatory. It should not be overloaded with options, comments and components. A clean design is required to avoid confusion and provide a frictionless experience.

ID	Requirement	Priority
1.3	Have a simple and intuitive UI	Medium
1.4	Support code fragment handling without additional tracing instructions	High
1.5	Provide convenient and non-intrusive help to explain how to use the application	Medium
1.6	Have a fast UI without waiting time	High

### 2.1.3 Target language

The target language refers to the code language in the code editor part of the UI. It is the coding language of code fragment provided or chose by the user and to be analysed by the application. As the application could be adapted to handle multiple programming languages, only one language is required to reduce the scope of the solution.



ID	Requirement	Priority
1.7	Target a simple and easy to learn language	High
1.8	Target a popular and widely used language	High
1.9	Support most common features of general coding language (eg. in contrast to non-general usage language like VHDL)	High
1.10	Target a language with good support for code analysis and code generation tools (eg. <a href="#">AST</a> tools)	High

## 2.2 Architectural requirements

The architectural requirements refer to the construction design of the application. They detail the constraints and the motivations for further development and improvement.

### 2.2.1 Extensibility

The project size of this application is meant to grow regularly and indefinitely. Mainly, the application requires to run different code fragments to represent as many features of a language as possible, which can lead to considerably large input domain. It need to be flexible enough for continuous improvements and to handle a large collection of code fragments. For example, the application should be extensible to support an additional coding language in future development. Also, the application should support a growing number of graphical visualizations to represent further algorithms, coding features and data structures.

ID	Requirement	Priority
2.1	Use popular, well supported and modern technologies	High
2.2	Support adding more coding language (eg. JavaScript, C, C#, Java)	Low
2.3	Support adding more custom graphical visualization (eg. graph, tree, bar, diagram, pie chart, linked list, grid)	Medium
2.4	Support adding a large number of code fragments	Low

### 2.2.2 Testability

Even a simple algorithm such as Bubble sort with an input of 10 numbers needs more than 200 steps and multiple states of variables. It does involve multiple language features such as loop, conditional, variable, array and expression, which increase significantly the number of execution paths and the complexity of the task. Unit and integration testing can be very useful to prevent errors and avoid application system failure. Also, [End-to-End \(E2E\)](#) tests are useful for testing the [UI](#) and to validate its interaction with the [Document Object Model \(DOM\)](#) components.

ID	Requirement	Priority
2.5	Include at least one test per feature listed in <a href="#">Subsection 2.3.1 Common programming concepts</a>	High
2.6	Be testable by integration tests to validate the correct interpretation of the code fragments	Medium
2.7	Have a <a href="#">UI</a> testable with <a href="#">E2E</a> tests	Medium

## 2.3 Functional requirements

This section lists all requirements that describe the expected functionalities of the solution. Also, the functional requirements detail the [UI](#) components.

### 2.3.1 Common programming concepts

In this section, a detail list of requirements describes the most popular features that the application should support. The system should recognize these features and be able to display them in a simple and explanatory layout.

ID	Requirement	Priority
3.1	Handle loop statements (eg. for, while)	High
3.2	Handle conditional statements (eg. if, else-if)	High
3.3	Handle operators (eg. assignment, comparison, arithmetic, logical, string)	High
3.4	Handle primitive type (eg. integer, string, boolean, character, decimal)	High
3.5	Handle primitive array collection	High
3.6	Handle advanced collections (eg. set, map, dictionary)	Medium
3.7	Handle comments	High
3.8	Handle function (eg. declaration, parameter, calls)	Medium
3.9	Handle function recursion	Medium
3.10	Handle custom data structure (eg. struct, class, object)	Medium
3.11	Handle variable scopes	Medium
3.12	Handle closure	Medium
3.13	Handle variable hiding / shadowing	Medium
3.14	Handle assignment expressions (eg. ++, + )	Medium

### 2.3.2 Set of basic code examples

The application should provide to the user a default list of various basic code fragments. These codes represent the most common features of a programming language, various type of algorithms and data structures.

ID	Requirement	Priority
3.15	Include at least one example for each main feature listed in Sub-section 2.3.1 <a href="#">Common programming concepts</a>	High
3.16	Include basic data structure operations (eg. Stack, Queues, Lists)	Medium
3.17	Include sorting algorithms (eg. Selection sort, Bubble sort, Merge sort, Quick sort, Shell sort)	High
3.18	Include indexing algorithms (eg. Binary Search Trees, Red-Black Trees, Hash Tables)	Low
3.19	Include graph algorithms (eg. Graph search, Dijkstra's, A*)	Low
3.20	Include dynamic programming algorithms (eg. Calculating nth Fibonacci number)	Low

### 2.3.3 Code edition

The code editor is the **UI** section where the code fragment will be inserted. The user should be able to clearly read, modify and work with the code.

ID	Requirement	Priority
3.21	Allow copy-paste of code fragment	High
3.22	Allow to modify any code fragment example already available	High
3.23	Highlight the active expression being executed in the code fragment	High
3.24	Display a syntax coloration of the corresponding coding language	High
3.25	Handle invalid code and display gutter-markers	Medium
3.26	Support basic auto-completion	Medium

### 2.3.4 Variable visualization

All variables are displayed and updated for a certain time during the execution of the code. Their values are shown along the name of the variable. Only active data is displayed and depends on the current step of the code execution. For example, if a variable is outside the scope of the current step, it should be disabled. Finally, each value should be updated for every step.

ID	Requirement	Priority
3.27	Display all variables name and value existing in the scope for a given step	High
3.28	Display variable of primitive type (eg. integer, string, boolean, character, decimal)	High
3.29	Display variable of type array and list	Medium
3.30	Display variable of custom data structure (eg. struct, class, object)	Medium
3.31	Display the updated value for each variable for a given step	High
3.32	Highlight variable and value when they are updated in a given step	High

### 2.3.5 Custom graphical visualization

This section defines the options for the graphical visualization. The bar diagram represents the values of variables such as arrays and lists. It updates the values every new step and can handle multiple visualizations at the same time. For example, a bar diagram can be displayed to help to understand how the Bubble sort algorithm is sorting the array. Also, for each code fragment example available, a preset of the variable to be shown in a diagram should be already defined. Alternatively, the user should have the option to choose the visualization he wants to see.

Some algorithms previously mentioned in [Requirement 3.18](#) and [3.19](#), would certainly benefit from additional graphic types such as graph, tree, linked list and grid. However, the effort to implement them would be too important to consider it a requirement with *High* priority.

ID	Requirement	Priority
3.33	Support the display of multiple custom graphical visualization simultaneously	Medium
3.34	Provide a way for the user to define which graphic visualization to display	Low
3.35	Have a default preset of graphic visualization per code fragment	Medium
3.36	Display the updated value for each custom graphical visualization for a given step	High
3.37	Highlight part of the custom graphical visualization that changes at a given step	High
3.38	Support the display of bar diagram	High
3.39	Support the display of tree diagram	Low
3.40	Support the display of graph diagram	Low
3.41	Support the display of memory layout (Stack and heap)	Low

### 2.3.6 Code execution navigation

This section gives to the user the full control over the *execution steps* of the code fragment. A playback toolbar contains all basic commands such as the *Play* button to start the code execution. The *Build* command is used to start the whole code analysis process and can be used every time a change appears in the code fragment. A slider widget should be available to navigate to any steps.

ID	Requirement	Priority
3.42	Provide a slider	High
3.43	Provide playback options (eg. play, forward, backward, jump step)	High
3.44	Display the current execution step and the total step count	High
3.45	Provide a <i>Build Visualization</i> button when the code fragment has changed	Medium

### 2.3.7 User interface feedback

Considering a reasonable scope, this application cannot handle all possible features. Many different languages with an almost infinite possibility of code and feature arrange-

ments make the task unrealistic. The application should instead handle unsupported features and communicate clearly to the user. Moreover, outstanding variable values and complexity of a code can lead to strange behavior and wrong graphical representations. Hence, limits should be established to avoid error and system failures.

ID	Requirement	Priority
3.46	Indicate unsupported features	Medium
3.47	Handle run time error and notify the user (eg. infinite loop, null pointer exception and reference error)	Medium
3.48	Handle outstanding variable values (eg. display of values 1, 2, 3000, 4 in a bar diagram)	Low
3.49	Limit code complexity to a specific number of steps or number of lines	Low

## 2.4 User interface

The following sections describe each part of the UI highlighted in the layout overviews from Figure 2.1.

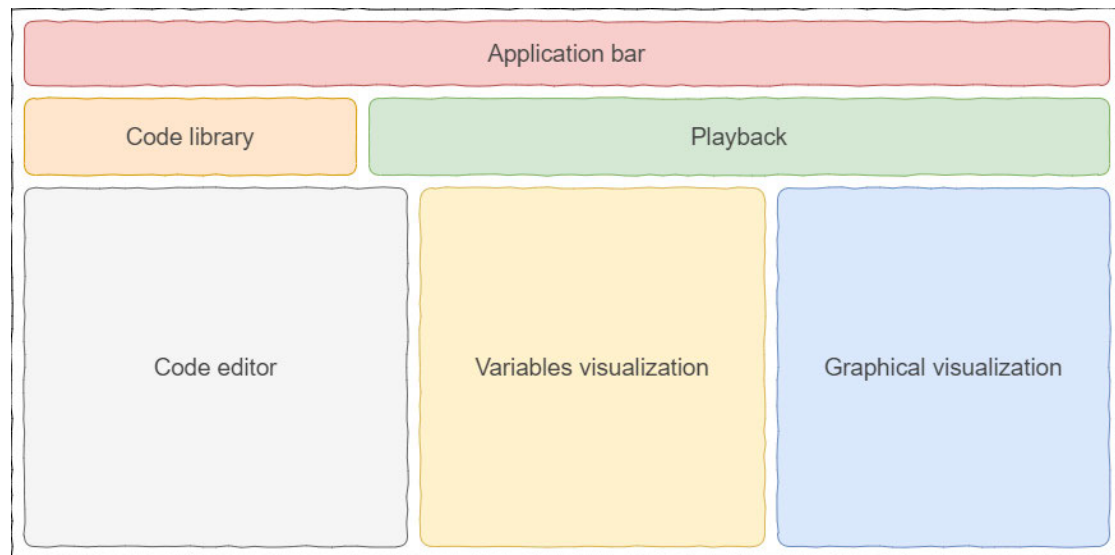


Figure 2.1: User interface layout overview.

### 2.4.1 Application bar

The application bar section displays the icon, the title and a short description of the application. It also holds option buttons on the right side.



Figure 2.2: Mockup of the application bar section.

The *About* option is there to explain the reason and purpose of this application. It also shares a link to the git repository project source code.

The settings enclose all options that will be needed in future development of the application such as the selection of the coding language.

The help option displays a popup with instructions on how to use the application. This way, it doesn't bother the user if help is not needed and fulfill [Requirement 1.5](#). To accommodate the [Requirement 3.46](#), another responsibility of the help section is to communicate to the user all unsupported features.

### 2.4.2 Code library

The code library is a collection of code fragments provided by the application. To satisfy the requirements from Subsection [2.3.2 Set of basic code examples](#), it contains a set of basic code fragment examples such as the Bubble sort algorithm or smaller code feature like a for loop. A drop-down list displays the name of all available code fragments. Once the selection is done, the list collapse and the name of the example appears on the box.

This choice of design only partially fulfill the [Requirement 2.4](#). Supporting a large number of code fragment implies a better structure, organisation of the code and a way to navigate easily through all samples. A drop-down list is a simpler solution, but would need adjustment later.



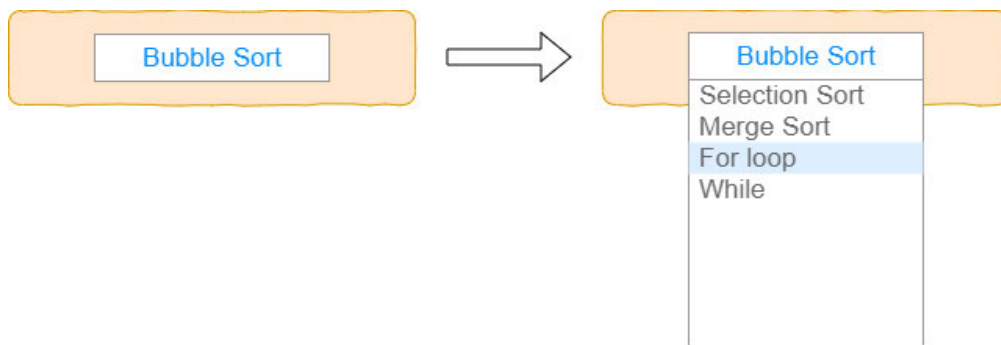


Figure 2.3: Mockup of the code library section.

### 2.4.3 Playback

All requirements of Subsection 2.3.6 *Code execution navigation* are taken into account. The *Playback* area displays a slider that gives the user the control on the navigation of the code execution. He can jump to any step or reach the end of the execution in one click using the slider.

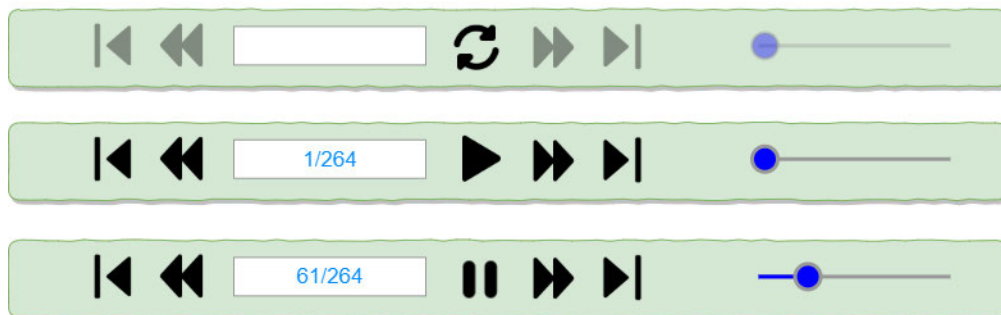


Figure 2.4: Mockup of three different states of the playback section.

The section also contains a counter to indicate the total number of steps and the current step progression of the code execution. Every new step or uses of the slider triggers an update of the current step.

The *Playback* area consist of buttons for the navigation through the code execution. The standard playback buttons provide a more granular way to navigate the code execution steps. The *Play* button goes automatically through the steps one by one at a specified speed. When the user presses the *Play* button, it toggles to a *Pause* button. The *Previous*

button rewinds to the previous step and the *Next* button advances to the next step. The *First* and the *Last* buttons jump to the respective step.

As soon as a modification appears in the code editor, the *Pause* button toggles to *Build Visualization* button. This button restarts the code analysis and generate the visualization, reset the current step to one and toggle back to the *Play* button.

### 2.4.4 Code editor

This area displays the code fragment example selected by the user and to be executed by the application. The code editor has a clean design and common features like line numbers and syntax coloration according to the target language. This code editor follows the requirements from Subsection 2.3.3 Code edition.

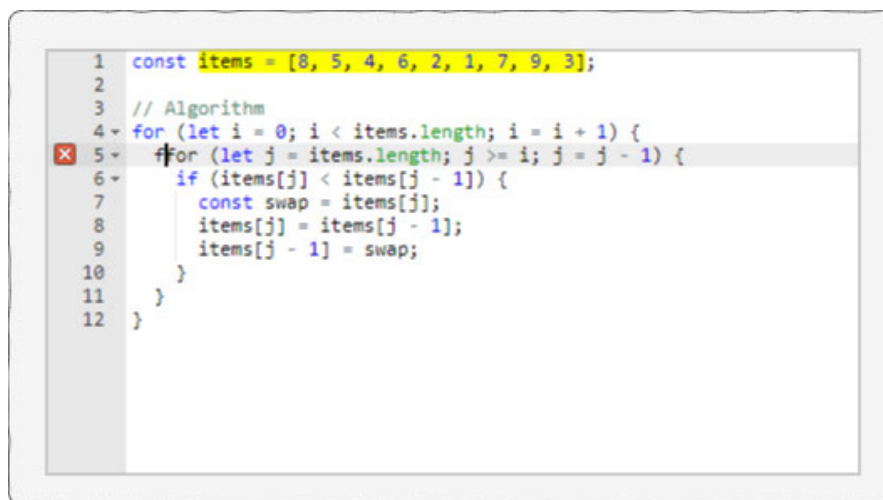


Figure 2.5: Mockup of the code editor section.

If an invalid code is inserted, a symbol corresponding to the error is displayed next to the error location until correction. For a runtime error or an unexpected behavior, a warning message appears to meet Requirement 3.47.

To satisfy the Requirement 1.4, at no point the application displays additional instructions to the initial code fragment. An example of cluttered code that is unwanted is shown in Appendix A. The screenshot shows how a simple *Bubble sort* algorithm can become difficult to read when cluttered by many tracings.

### 2.4.5 Variable visualization

Keeping in mind that this section is a support for reading a code, the design must stay as simple as possible. Contrary to other applications that try to reproduce the memory of a computer, this area's only concern is to expose a simple list of variables with their values. It abstracts the complexity of the memory management such as the local and global scope, stack and heap. The display does not show all variables in the code fragment at the same time, but reveals only the variable existing in the current execution scope, refreshing their values at every new step. As expected, this section grows and shrinks for every variable creation or deletion.

The Figure 2.6 shows the variable's name and values of a Bubble sort code at a specific moment of the execution code.

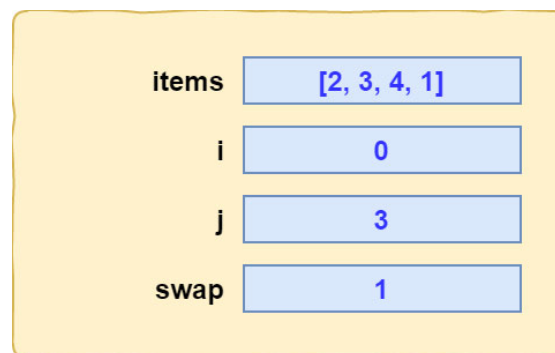


Figure 2.6: Mockup of the variables visualization section.

More examples of variable visualizations are displayed in [Appendix B](#) and give a general idea of how to handle the requirements of [Subsection 2.3.2 Set of basic code examples](#). Each represents a snapshot of a specific step during the code execution and illustrate different coding features.

While this type of visualization focuses on simplicity, it reveals some limitation for more complex features. For example, an object that contains references to itself or to other properties is not adequately represented with this visualization. Such cases would require a more advanced visualization such as the one illustrated in [Appendix C](#), but is not covered in the current effort.

### 2.4.6 Graphical visualization

This section illustrates a graphical visualization to complement the variables visualization for more complex types of variables. For example, sorting algorithms can be better understood if the values are represented by a bar diagram. As shown in Figure 2.7, the user has the option to add a variable of his choice and associate it to a type of graph that he wants to see. For a code fragment provided by the application, a preset set the default graphical visualization to display. This feature is optional and extend the functionality of the application to meet Requirement 3.34 and Requirement 3.35. Ultimately, various types of graph such as trees, histogram, chart and diagram will be available in the application, but to fulfill Requirement 3.38, only the bar diagram type is supported. Moreover, multiple variables can be displayed simultaneously. They are updated and partially highlighted at every step.



Figure 2.7: Mockup of the graphical visualization section.

Finally, gathering every parts described in this section produce the final result of the UI shown in Figure 2.8.

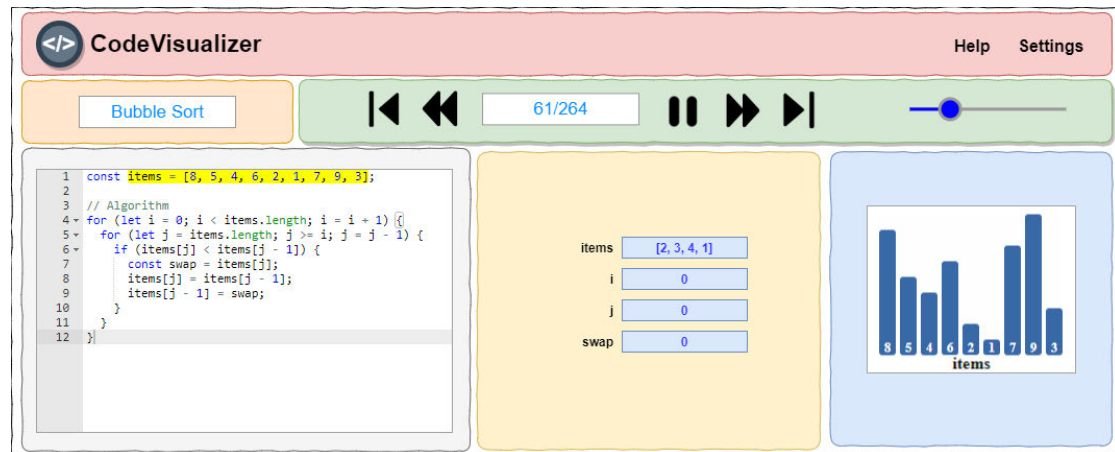


Figure 2.8: User interface overview.

## 2.5 Approaches for code execution visualization

There are different ways to realise a visual from a piece of code. Based on similar applications found on the web, four approaches are worth considerations. The Table 2.1 compares them based on important properties. The following sections discuss the potentials and the possible issues for each of the approaches.

Approaches vs criteria	Scripted animation	Tracing instruction	Debugger supervision	Code transformation
Developer can add code fragment easily	Limited	Good	Very good ✓	Very good ✓
User can add code fragment easily	No	Limited	Very good ✓	Very good ✓
Scalability of language feature support	Very good ✓	Very good ✓	Good	Limited
Code highlight support	Very good ✓	Limited	Limited	Very good ✓

Table 2.1: Comparison of approaches to visualize code execution.

### 2.5.1 Scripted animation

A naive brute force way to think about creating a visual from a code is simply to hard-code it. With this approach, visualizations are custom-scripted for each code fragment. The majority of the platforms currently on the web use this option.

Since the code is static and cannot be altered at any point by the user, it is easier to test and prevent almost any error and implementation complexity. Also, there is no barrier to adapt the visual to any code language.

However, the user cannot paste his own code neither modify the code fragment. The developer needs an extensive amount of time to produce each example and users are limited by those examples. Yet, the potential quality of the visual can be impressively good and code highlight can be set as desired. A good website that seems to implement visualizations for algorithms in such a way is [visualgo.net](http://visualgo.net)[55].

### 2.5.2 Tracing instruction

Algorithm Visualizer[4] is the name of a similar interactive online platform to visualize algorithms. However, it adopts a different approach that allows more interactivity and options for the user. The application uses custom tracer libraries to derive the visual features from the code fragment. The developer provides `tracing instructions` to be added in the code fragment. Which means that the initial code contains additional instructions and from this combination, the system extracts the necessary data for the visualization.

The platform is compatible with C++, Java and JavaScript, and more languages are possible. A developer can create animated visual more freely thanks to the common `tracing instructions`, but each type of graphical visualization needs its own specific instructions.

This approach is a major improvement compared to the scripted animation style in term of flexibility for the user, since he can directly modify code inside the code editor to see its graphical representation.

On the down side, the tracer instructions produce noise in the reading of the code fragment. The code editor section shows the full code including all tracers functions. It makes the logic of the initial algorithm code and the variable update harder to follow.

Moreover, the user cannot simply copy-paste his own code, but need to add some tracer commands in order to trigger the visualization. In other words, the user need to learn how to use the custom library of the application which can overwhelm a novice and limit the usability of the application.

At last, precise highlight in the code is not available since no instructions can provide the exact location of the code section currently active.

### 2.5.3 Debugger supervision

Another very interesting approach is used by Python Tutor[38]. The main idea is that a debugger executes the algorithm and outputs the relevant frame information for each code step. In order to display a visual from a code fragment, this application uses an existing compiler and a back-end server. For example, with Javascript, the application uses the open source engine of Google called V8[53] to execute and extract all data from the code fragment. With Python language, it uses DBD[18] module to handle the execution via the debugger. With Java, it works directly with the Java Development Kit (JDK). And finally, it bases the dynamic analysis of C and C++ code on a tool called Valgrind[54]. The uses of these compilers and debuggers is a time saver for the development of new languages.

This platform is compatible with different languages such as C, C++, Java, Python, JavaScript, TypeScript and Ruby. Even if the application grants a good scalability to add more language, it is bound by the amount of work required per language implemented, but not per features of the languages. Nevertheless, troublesome language features like the concept of pointers and scopes are handled surprisingly well. As illustrated in the Figure 2.9, the visual displays the swap of two variables using pointers from a C code. The display also separates the stack and heap memory for the variable.

As a first disadvantage, the compile time for getting the result is significant since it needs to send the code to a server, run it on the back-end side and return it after extracting all information necessary for the execution. The application needs around 10 to 20 seconds to execute a simple code fragment of 20 lines.

The visual quality is limited to the information obtained from the debugger, which mean that the program could lack data in order to highlight the code fragment at the good place.

The screenshot displays a C program in Python Tutor. The code is as follows:

```

C (gcc 4.8, C11)
EXPERIMENTAL! known limitations
1 #include <stdio.h>
2
3 void SwapInt(int* number1, int* number2) {
4     int temporary;
5     temporary = *number1;
6     *number1 = *number2;
7     *number2 = temporary;
8 }
9
10 int main() {
11     int number = 1;
12     int number2 = 2;
13     SwapInt(&number, &number2);
14
15     printf("%d", number);
16     printf("%d", number2);
17
18     return 0;
19 }

```

Below the code is a legend: a green arrow indicates the line that just executed (line 7), and a red arrow indicates the next line to execute (line 8). A progress bar shows the current step is 9 of 12. Navigation buttons include '<< First', '< Prev', 'Next >', and 'Last >>'. A link for 'Customize visualization (NEW!)' is also present.

On the right, a memory diagram shows the Stack and Heap. The Stack contains:

- main**:
  - number: int 2
  - number2: int 1
- SwapInt**:
  - number1: pointer
  - number2: pointer
  - temporary: int 1

Arrows indicate that the pointers in the SwapInt frame point to the memory locations of number1 and number2 in the main frame.

Figure 2.9: Screenshot of an example in Python Tutor.

### 2.5.4 Code transformation

The last approach presents a new idea which is the selected approach for this application. Yet, previous research shows that no other application is using this method. It uses the transformation of an *AST* to dynamically display a visual from the executed code.

The workflow involves six main steps that require to parse, modify, generate and execute code. More explanations are detailed in Subsection 4.3.2 *Process data flow*.

1. Get a code fragment from user.
2. Convert the code fragment into an *AST*.
3. Augment the *AST* with inspection method calls.
4. Use augmented syntax tree to generate an augmented version of the code fragment.
5. Execute the augmented code fragment to generate execution steps data.



6. Use the [execution steps](#) data to visualize the code fragment execution.

Each feature listed in the requirement Subsection 2.3.1 [Common programming concepts](#) need to be implemented only once for the [AST](#) to automatically transform its code. The application must cover all existing features of a language to support it. Consequently, it makes it harder to support additional languages and fulfill [Requirement 2.2](#). Additional language would need major adaptation and different tools.

Amongst all advantages, this approach makes it very simple for both the developer and the user to write code fragment. Also, the [AST](#) contains the precise locations of each part of the code. The application uses these location data to highlight directly in the code, and make it easier for the user to follow the [execution steps](#).

Considering the granular and custom handling of each language feature, the potential for the quality of the visualization and good user experience seems to be maximized with this approach.

## 3 Theory

The theory chapter clarifies technical concepts required for a better understanding of the application's structure and foundation. The first part describes the [AST](#) anatomy and functionalities. It is necessary to understand the syntax analysis process since it is the approach selected in [Subsection 2.5.4 Code transformation](#) for the application to transform the code. The section also depicts some useful use cases of the [AST](#).

The second part explains some specific advanced programming features. These features introduce some challenges in the development when it comes to visually illustrate them. They are portrayed with simple examples.

### 3.1 Abstract syntax tree (AST)

In order to understand what is an [AST](#), let's examine each of its term separately.

In computer science, a tree is a collection of data organized in a nonlinear data structure that includes a root value at the top and subtrees of linked nodes. It can represent an ordered collection of nodes, where each child node has at most one parent node.

The syntax is simply a set of rules used by a program to define a combination of symbols to represent a language or data. By opposition to the semantic concept that reflect the meaning of the code, the syntax only validates the grammar of the language. From this perspective, a syntax tree or also called a [Concrete Syntax Tree \(CST\)](#) is a strict grammatical representation of the source code in a treelike form. It shows how a parser understands the code.

Contrary to a [CST](#), an [AST](#) is used to analyse the code and thus does not need syntactic details such as semicolons, parentheses and commas. The concrete syntax tree has generally more details and is more complex to read. In comparison, an [AST](#) will be smaller and simpler because it abstracts all unnecessary information.

As demonstrated in this work, it is possible to use an [AST](#) to inspect and modify a source code. The magic comes with a process flow of three main operations explained in the following sections: *parse*, *traverse* and *generate*.

#### 3.1.1 Code parsing

The process of parsing is simply explained by detailing the first two phases done by a compiler. The first phase consists of a lexical analyzer scanning a source code and splitting it in atomic parts called lexemes. Then, a lexer, also called tokenizer, takes these lexemes, removes unnecessary symbols such as space characters and comments and converts them into tokens.

In the syntax analyzer phase, the parser creates a syntax tree using the list of tokens created in the previous phase.[33] The parser's job can be either to create a [CST](#) if all tokens are used, or to create an [AST](#) if some unnecessary tokens are omitted. It is important to mention that a parser will always generate an [AST](#), but the [CST](#) is optional and depend on the parser usage. In the end, an [AST](#) is a subset of its [CST](#) counterpart and results to a simplified tree as it abstracts away all superfluous leaf nodes.

#### 3.1.2 Anatomy of an AST

Before undertaking any transformation on an [AST](#), it is necessary to understand how it is constructed. The tree is made of linked nodes that has exactly one parent node each, except for the root node. Also, they can have sibling and child nodes. For example, the root node of an [AST](#) with Babel will be named *Program* and will consist of all top statements of the program. The [Figure 3.1](#) displays an [AST](#) of a simple array declaration.

```
1 const items = [8, 5];
```

Listing 3.1: Simple array declaration.

```
- File {
  errors: [ ]
  - program: Program {
    sourceType: "module"
    - body: [
      - VariableDeclaration {
        - declarations: [
          - VariableDeclarator {
            - id: Identifier {
              name: "items"
            }
            - init: ArrayExpression {
              - elements: [
                + NumericLiteral {extra, value}
                - NumericLiteral {
                  + extra: {rawValue, raw}
                  value: 5
                }
              ]
            }
          }
        ]
        kind: "const"
      }
    ]
    directives: [ ]
  }
  comments: [ ]
}
```

Figure 3.1: AST of an array declaration constructed by the JavaScript parser Babylon7 used by Babel.

All nodes of the tree have a type name in relation to their functionality and these names are defined by the parser. Each type of node has a specific list of additional properties. For example, with the parser Babel, a node that represents a declaration of variable has the name *VariableDeclarator* and the property *Identifier*. There exist many more types<sup>[10]</sup> in order to fully cover the complexity of one language.

### 3.1.3 AST traversal

Traversing an *AST* is the second main operation and refers to the exploration of the *AST*. It is now possible to navigate, analyse and potentially extract data from the tree. To explore an *AST*, each node is visited in order, starting with the root node.

To manually find a specific node, the path must be known. Some tools such as [AST Explorer](#)[8] exist and can help to see directly the [AST](#) from a source code. It is also very helpful to find the path to access a node.

#### 3.1.4 [AST transformation and generation](#)

The manipulation of the [AST](#) is not mandatory, but it is very important to understand in the context of the current work since it is the heart of the application process. The main idea is to transform an [AST](#) into another [AST](#) by adding, replacing or removing nodes. This manipulation can be complex since it requires a good knowledge of the [AST](#) structure, the types and properties available to use.

This transformation leads to a change of the source code, such as changing the behavior of a method or the output value. The transformation of an [AST](#) can be used to create a custom JavaScript syntax. Once an [AST](#) has been modified, the usual next step is to convert it back to code by a process called generate.

## 3.2 [Advanced programming concepts](#)

To explain a simple code fragment, it can be sufficient to visually represent a list of variables. However, this approach has its limits when it comes to explain more advanced programming concepts. The challenge resides in the balance between the simplicity and the more accurate explanation of the memory management. These concepts and their challenge for the visualization are explained in this section.

### 3.2.1 [Memory management](#)

When visually listing variables involved in a source code, it is important to consider how a computer is managing the memory. Some data is allocated on the stack, other on the heap. For advance concepts such as the closure concept, it is needed to give a better representation of the code logic and this involves a good understanding of the stack and the heap.

A stack operates in First-In-Last-Out (FILO) mode. The memory from the stack cannot be fragmented or resized. Thus, it is usually much faster, because it has a linear data

structure. The stack memory can only access the local variables and is safer than the heap.

The heap is a large region where data is stored in a seemingly random way. It allocates dynamic data, allows access to global variables and the variables can be resized. Consequently, the heap requires manual variable de-allocation, or more complex memory garbage collection system, hence is considered less safe and less performant than the stack.

#### 3.2.2 Variable scope

For a variable to be reachable, it needs to be used in its active region where it is located and only exists within the block where it is declared. If the variable is global, then it can be used anywhere and by every function. If it is a local variable, then the variable will be deleted as the code execution exits its scope block.

To sum up, if a list of variables must be illustrated, some variables need to disappear when they become inactive. As shown in the [Table B.1](#) of the [Appendix B](#), the visualization displays the variables as the execution of the code is at the line 17. It displays in blue the active variables and in red the variables that do not exist in the current scope.

#### 3.2.3 Variable hiding

Variable hiding happens when a variable is defined in a scope that is inside another scope block, and a variable with the same name already exists in a parent scope. While this child scope is active, the new variable overrides the previous variable from the parent scope which cannot be referenced by any mean. This can lead to error during the execution of the code without warning. The [Table B.4](#) from [Appendix B](#) shows that the variable *a* can print to the console different values, depending only on the scope of the function that uses the variable.

#### 3.2.4 Closure

When a variable is declared in the scope of a function, it exists only while the function is active, and is erased from the memory when the code execution exits the function. Closure is a technique that allows to have persistent local variable scopes. It allows to

keep in memory variables, even outside of their scope. The binding of the variable is done as long as the function persists.

JavaScript is one of the languages that support the closure concept and thus, the visualization of the variable with closure should be kept active, even after the execution of the code has finished the block of scope of this variable. As shown in the [Table B.5](#) of the [Appendix B](#), the display doesn't represent well the value of the inner variable. A more complex visual of the same code variable is done in the [Figure C.1](#) and [Figure C.2](#) of the [Appendix C](#). It shows an attempt to display a closer representation of the memory storage.

## 4 Design

This section describes the design choices leading to a candidate solution. The first section refers to the choice of the tools and libraries. A comparison of different technologies based on previous researches explains their functionalities and uses. The second section of the design explores various pros and cons of the [target language](#). This language refers to the code fragments that will be analysed by the application. Finally, the last section brings up the application architecture and how all the [modules](#) interact together.

Before diving into these specific parts, a use case diagram shown in [Figure 4.1](#) illustrates a situation between the user and the application.



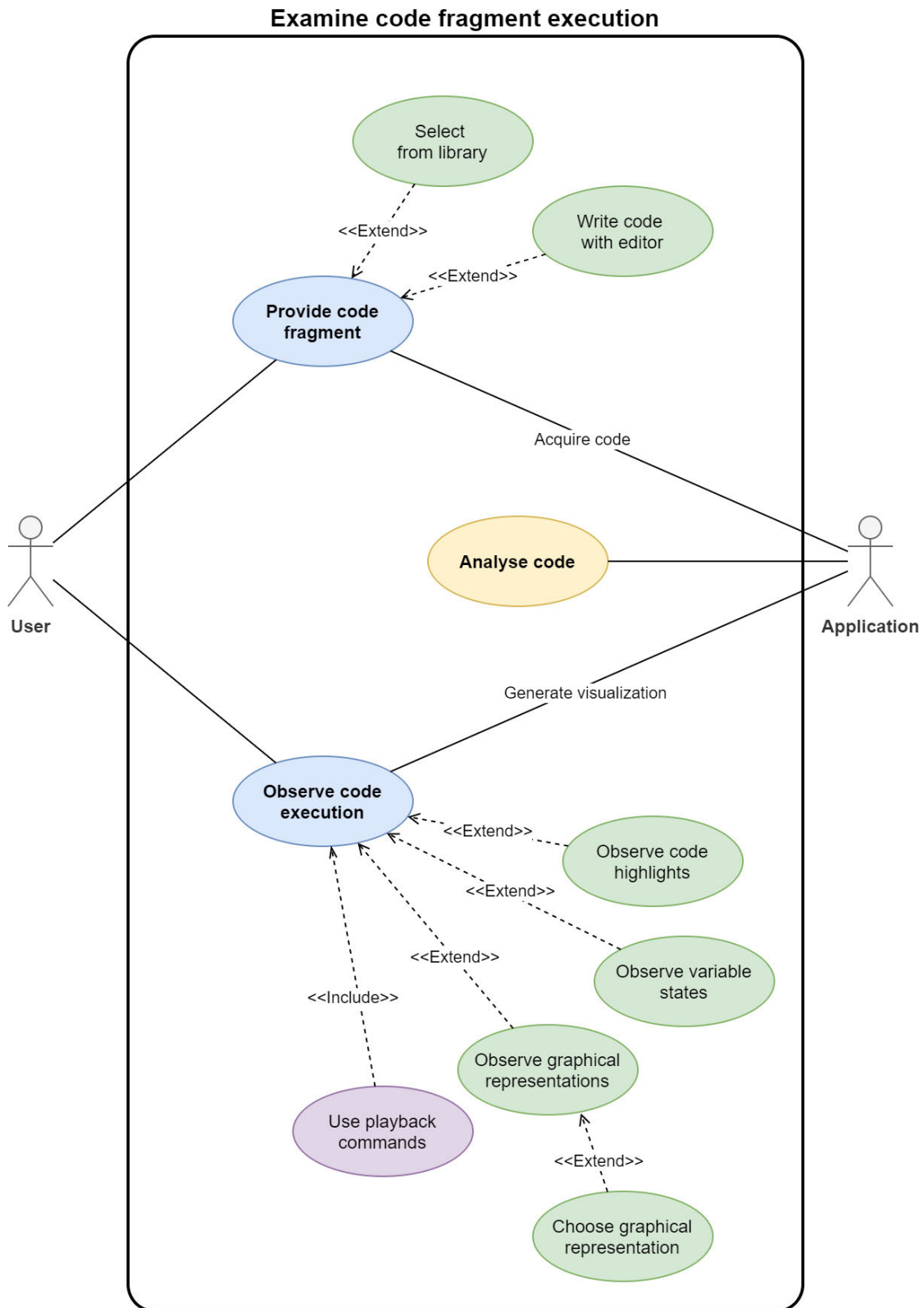


Figure 4.1: Use case diagram of a user examining a code fragment execution.

The main functionality of the application is to analyse a code fragment and display the code execution with graphical visualization. From the user perspective, two actions are possible. First, the user can select a code fragment provided by the application or write a code in the code editor. Writing can consist of modifying a code fragment provided by the application, adding from scratch a new code or copy-pasting a source code. Second, the user can observe the code execution and control the workflow with the playback commands.

On the application side, three functions are processed in order. The application gets the code fragment, transforms it and generates the visualization.

### 4.1 Technologies

The choice of technologies is based on the [Requirement 2.1](#), which relate to popular, well supported and modern technologies. In contrast to the [Section 4.2 Target language](#) which focuses on the [target language](#) of the code fragment examined by the user, this part of the design refers to the code language and tools used to build the application.

#### 4.1.1 Web platform

The selected platform for this application is the web. Assuming that most people have an Internet connection and an operational browser, the web is the most accessible platform for a computer. As specified in [Requirement 1.1](#), it does not require the user to install or download any software or plugin. To follow [Requirement 1.2](#), it is also possible to adapt the application for different browser versions and [Operating Systems \(OS\)](#).

The choice consequently involves [HyperText Markup Language \(HTML\)](#), [Cascading Style Sheets \(CSS\)](#) and [JavaScript](#) technologies. While other languages have been considered, JavaScript is the most widely used for front-end web application. Popular websites such as [Google.com](#), [Facebook.com](#), [YouTube.com](#), [Amazon.com](#) and many more use JavaScript[34]. Furthermore, some of the most recent statistics from [GitHub](#) and [Stack Overflow](#) show that JavaScript is one of the most popular programming languages[44].

Finally, [TypeScript](#) is also considered since it helps to write better, safer and more organized code and is more scalable. The main advantages of TypeScript in comparison with JavaScript is that it handles additional useful features like the static typing and

generics. To limit the scope of the current project, TypeScript will not be used, but reconsidered in the future.

### 4.1.2 User interface library

Without proper tools and methodologies, the UI can be challenging to create. There exist many libraries that can help to build and organize the view code. They provide a component based structure that wraps part of the UI into dynamic components. Therefore, the developers can manage and test components more easily and avoid to write hard to maintain vanilla JavaScript and HTML code.

Also, the operations done on the DOM are slow. Manual changes of the document structures, style and content usually do not lead to optimized code. Libraries can minimize the operations done on the DOM and improve the performances of the rendering, which contribute to fulfill the Requirement 1.6.

Among the best and most popular tools to be used alongside with JavaScript, three libraries stand out: Angular, React and Vue.[7]

Angular[5] was the first among these Web frameworks and was released in 2010 by Google. React[42] was released in 2013 by the Facebook team. The youngest of all three frameworks is Vue[57] and was released in 2014 by the community.

The table Table 4.1 and the following sections compare these tools based on different criteria and confirm the choice of selecting React as the main UI library for this application.

Criteria	Angular	React	Vue
Feature set vs flexibility[1]	Opinionated	Flexible ✓	Flexible ✓
Learning Curve[15]	Steep	Smooth ✓	Smooth ✓
Popularity	Good	Very good ✓	Good

Table 4.1: Comparison of popular web frameworks relative to custom criteria.

### Feature set vs flexibility

Angular is more than just a UI library, it is a heavy web framework that contains a complete solution, with lots of built-in features. Its download size of 500 Kilobyte (KB) is massive compared to React with 100 KB and Vue with 80 KB. Hence, the applications developed with Angular usually need more memory space and time to download.

Angular is opinionated, constraining the development in a specific direction. It provides lots of features, some are completely optional, but others, such as injectables and module structures, forces the developer to use them even if not needed. It uses the [Model-View-Controller \(MVC\)](#) structure concepts and specific directives for its components, structures and attributes. It offers considerably less flexibility than the other alternatives.

On that front, both React and Vue are very similar, they can be customized by adding third-party tools for advance features and to scale up as the project grows.[58] They provide a flexible way of organizing components, but can also introduce a state management library (Redux and Vuex).

### Learning Curve

Learning Angular is harder than its competitors because it is a large framework with many features. Considering the relatively small scope of this application, Angular does not align well with the requirements.

Vue uses its own pre-processors rather than normal CSS. It has its own template directives and custom options for the component that need to be learned and memorized.

For a minimal usage, React has a very small [Application Programming Interface \(API\)](#) surface that allows to use its functionalities with a least amount of effort.[47] It uses a syntax extension of JavaScript called [JavaScript Extensible Markup Language \(JSX\)](#) that allows to write markup inside a JavaScript code to produce React elements. React uses props to map and render the component's properties while Vue uses additional options such as slot content, a built-in template loop function.[19] The concept encouraged by both React and Vue is the [Component-Based Architecture \(CBA\)](#) and differs from the more conventional [MVC](#) architecture by offering a less coupled and easy way to reuse existing components.[12] Also, a fairly new feature in React called *hooks* allows functional

components to have a state and be updated when modifications occur. While the [UI](#) and the behavior of the components are separated in [Vue](#), it is combined in [React](#).

To summarize, [React](#) follows more closely the [JavaScript](#) and [HTML](#) languages than its competitors. It is the one with the smallest [API](#) surface for minimal usage and suits better the needs of this project.

### Popularity

Since popularity is debatable, multiple metrics should be considered. Both [Vue](#) and [React](#) have high popularity on [GitHub](#) such as the amount of stars and forks they have. [React](#) stands out for the trends in the job market, for the number of [Node Package Manager \(NPM\)](#) download count and for its [GitHub](#) contributors.[32][43] Furthermore, based on the recent [StackOverflow](#) survey of nearly 65,000 developers, [React](#) is the most popular, loved and wanted among all three frameworks.[49]

In conclusion, even if [Vue](#) would certainly fulfill the requirements of this project, [React](#) is chosen for its additional maturity, simplicity, popularity among professionals and because it is developed by the solid company [Facebook](#).

#### 4.1.3 User interface style library

This application needs a simple and intuitive [UI](#). Many [CSS](#) frameworks and libraries can facilitate the process of creating user-friendly, simple and stylish user interface. Furthermore, they can help to develop sophisticated design as the application grows and becomes more complex. Among the best and most popular tools, [Bootstrap](#) is the most widely used and can provide good quality front-end components to a web page. With *React Bootstrap*, developers can access thousands of different themes. Unfortunately, this framework is heavy and can encumber the application with functionalities that are not necessary.

There are many other options of [CSS](#) libraries that would certainly fulfill the needs of the current application. Also, considering the choice of [React](#) as the main [UI](#) library, the [CSS](#) library should go along it pretty well.

[Google](#) has identified a set of guidelines for the visual design of [UI](#) called [Material design](#)[27]. They improve greatly the quality of a [UI](#) and following these guidelines aligns

well with [Requirement 1.3](#). These concepts can be better implemented with the library Material-UI[28], which includes React components. It also offers many optional themes, including Google's theme. Finally, this library is smaller and have a good support from the community. Thus, Material-UI is the selected style library for this application.

### 4.1.4 Open source

As mentioned previously, the application mainly serves academic purposes. The main goal is to help learning code or give a simple tool to analyse a code. The product need to be accessible, free and available. The Open-source model shares these requirements and offers many more advantages. Mainly, the capacity of maintenance and the support of the community. Also, the code get fully visible to everyone, hence the solution get better security, robustness and the code is much more tested. In summary, delivering the application as an open-source software will allow the community to contribute to further development and enhance the solution for different purposes.

### 4.1.5 Text editor library

ACE editor[2] is one of the most popular and used code editor for the Web[22]. For instance, it powers Wikipedia and Khan Academy. It is written in JavaScript and supports most common source code editor features such as copy-paste, syntax coloration, code highlight, line numbers, auto-completion and notifications for coding errors via ESLint[20].

There are other suitable code editors available such as CodeMirror. The final decision to use ACE is arbitrarily based on its popularity and its capacity to support all requirements from [Subsection 2.3.3 Code edition](#).

### 4.1.6 Test libraries

The application needs different technologies for different type of tests with JavaScript code.

For unit and integration tests, Jest[23] is the most popular testing framework to use with JavaScript. It has a clever parallel testing, is simple to install and have a good compatibility with React and Node.js. Compared to other similar libraries like Mocha

or Jasmine, Jest offers more functionalities. For example, Jest is a test runner like Mocha[29], it executes the tests and gives a summary of the results. It is also an assertion library like Chai[13] that defines the testing logic and the conditions of a test. It is open source, has a great community support and Facebook recommends it. Hence, Jest is a great choice to fulfill both [Requirement 2.5](#) and [Requirement 2.6](#).

For [E2E](#) test, Puppeteer[35] uses a headless browser, which simulates browser interaction. It can mimic commands that the user would do on a web page and test the application with the results.

### 4.1.7 Module bundler and package manager

The more an application grows, the more code and files are involved. Either one JavaScript file contains all the code or multiple scripts contain each part of the code. Both options bring problems. It is difficult to work with one big file, hard to maintain, prone to bugs and reading from top to bottom is not efficient. The problem with the second approach is that all files need to be included in a specific order and this can be challenging if there are many dependencies. As a solution, a package manager can bundle all scripts and manage all dependencies without regard to the size of the code. It solves the problem of scope and readability as the code is kept in small [modules](#).

Webpack[59] is by far the most popular and complete JavaScript [modules](#) bundler and task runner. It takes scripts with their dependencies and packs them into few bundle assets such as [CSS](#), [PNG](#), [JPG](#) and JavaScript files.

One way to install Webpack is with [NPM](#)[31]. [NPM](#) is considered the world's largest software registry. For the JavaScript runtime environment [Node.js](#)[30], it is the default package manager. It acquires libraries and frameworks in a similar way that [Advanced Package Tool \(APT\)](#) allows to acquire applications for Linux.

Configuring Webpack to use the library [Babel](#) allows the application to support different versions of browsers. [Babel-loader](#)'s main role is to transform the source code to convert into an older compatible JavaScript code such as [ECMAScript \(ES\)5](#), [ES3](#), and more. Moreover, [Babel](#) is also used to convert the [JSX](#) syntax into JavaScript code, which is convenient when working with [React](#).

## 4.2 Target language

The choice of the **target language** is decisive because it is the coding language that the user will interact with. The decision of the **target language** depends on the language features, the tools available and other properties. Based on [Requirement 1.8](#), an evaluation of the most popular languages reveals their pros and cons in the current context. As a result, this comparison leads to a specific **target language** and consequently, compatible tools and libraries.

### 4.2.1 Language comparison

The candidate language selected for the comparison are based on the most popular programming languages as of 2020 Q1. Not all authors agree on the level of popularity and how to rank them. For example, the RedMonk.com[44] website extracts language rankings from GitHub and Stack Overflow, while the Tiobe.com[51] index bases its data from Google, Bings and Yahoo popular searches, the number of skilled software developers in the world and courses offers.

The [Table 4.2](#) summarizes relevant comparison criteria of the intersection of the top 10 languages from RedMonk.com and Tiobe.com. The following sections reveal details for each criteria.

Criteria	Complexity	Features support	Code analysis tool	Can run in Browser
C	Low ✓	Few	Moderate	No
JavaScript	Low ✓	Many ✓	Very good ✓	Native ✓
C#	High	Many ✓	Very good ✓	No
Java	Medium	Many ✓	Limited	No
C++	High	Many ✓	Limited	No
Python	Low	Many ✓	Good	Possible[39]

Table 4.2: Comparison of popular programming languages relative to custom criteria.



### Complexity

Two main reasons makes the complexity an important property. First, the application needs to be simple, fast and easy to develop. Second, the users likely to use this application are expected to have a low level of knowledge and experience in coding. It is important to find out which language is the most suitable for beginner and is a good entry for learning programming concepts.

Initially, the best language to meet the [Requirement 1.7](#) should be the one that is the most neutral. Many books and online tutorials use a flavour of pseudocode to describe code. It is an informal high-level code that details with natural language the operating principles of a code fragment. It abstracts the implementation details and make it easier for people to read. However, pseudocode is not a suitable solution in this context since it is not standard and by definition, cannot be built or executed on any platform.

All search results show that people use different metrics to evaluate the complexity of a code. The number of reserved words in a language is one of the way to measures it[26]. The [Table 4.2](#) shows a ranking level of complexity from low to high for each language. The ideal result is low level and represents a number of keywords under 50. The bigger the number is, the greater the size of the language is. For example, C# leads with a score of 102 reserved words and C++ closely follows with 93. The remaining languages show a very low number of keywords, hence C, JavaScript and Python are among the list, the most simple languages.

### Features support

The previous section reveals that it is better if the language is simple. However, following the product [Requirement 1.9](#), the [target language](#) also needs to support the most common features of general coding languages. One way to compare languages is to count the number of paradigms they support such as imperative, object oriented, functional, procedural, generic, reflective and event driven[25].

Here, the desired languages are the ones with many supported features. All languages on the list fulfill this requirement except for the C language, which supports only three of the paradigms from the list.

### Code analysis tool

The central feature of this application is to analyse a code fragment, hence the language selected needs to come with good tools and libraries to support this functionality as of [Requirement 1.10](#). The technologies to parse a code or to transform it into an AST are not always easily available for all languages. For example, libraries like `cppAST`[\[16\]](#) for C++ or `Spoon`[\[48\]](#) and `JavaParser`[\[21\]](#) for Java are not so popular and the support for these libraries seems limited.

Some tools are compatible with multiple languages such as `Clang`[\[14\]](#) for all C language family and `ANTLR`[\[6\]](#) for Python, C# and JavaScript. They are very interesting since they can handle more than one language, but they do not offer all the features required for this application. For example, ANTLR do not include the code compilation. Instead, developers mostly use it to create parsers.

For C code, a parser called `Pycparser`[\[36\]](#) is developed in Python and mostly used by the C Foreign Function Interface (CFFI) for Python. Its functionalities appear to be very specific and additional libraries would be needed if used.

Python includes a built-in code compiler and parsing tools to generate AST[\[37\]](#). This popular library has many contributors and offers many features that could fulfill the requirements.

For C#, `Roslyn`[\[46\]](#) has many features to offer for code analysis. It is an open source .Net compiler platform developed by the .NET Foundation and has several contributors. It is a complete solution that could meet the requirements. No other alternative for this language is as popular as Roslyn.

Compared to all previous tools mentioned, `Babel`[\[9\]](#) is the most complete tool and is compatible with JavaScript. It has many purposes and it is mostly used for converting new JavaScript code into a previous version for compatibility with older browsers and environments. With many contributors and being open source, Babel has a complete set of features for code analysis that are constantly updated. Many other libraries are also possible to use with JavaScript like `Acorn`[\[3\]](#), but mostly not as complete as Babel.

### Can run in a browser

To meet the [Requirement 1.6](#) to have a fast UI without waiting time, language that can run on browser are favored. The reason is that compiled languages need a server to run the code, which mean waiting time for the user. The Web browser runs the whole application without the help of any server or data transfer and the result is instantaneous. In that regard, JavaScript is apparently the best options since it is the main interpreted language used for front-end Web development.

It's worth mentioning that Python could also be a possible alternative. There are different ways to run Python on the web and could reach the same result as JavaScript, but Python is mostly used for back-end purposes.

### 4.2.2 JavaScript

JavaScript is often considered an easy language to learn and is often the entry coding language for many students in information technology field. It is also criticized for its numerous pitfalls, however most of them can be avoided with proper guidelines and development tools[17]. Among all popular languages, JavaScript is the one that stands for the technologies available, the contributors and the information surrounding the language on the web. Finally, based on the result of the [Subsection 4.2.1 Language comparison](#), JavaScript is the best choice of language to suit the application requirements.

### 4.2.3 Babel

This section defines the toolchain selected to parse and modify the code fragment with the help of [AST](#). It is the heart of this application and the application depends heavily on it. As mentioned in [Section 2.5 Approaches for code execution visualization](#), the selected approach involves parsing, modifying and generating code.

The best library to fulfill all the functionalities required is Babel. It is by far the most popular tool for source code transformations. As indicated in the [Subsection 4.1.7 Module bundler and package manager](#), the application already uses Babel at build time as a development tool. Nevertheless, there are some alternatives to Babel that are worth mentioning.

Sucrase[50] offers a super-fast development build compared to Babel. The library is a subset of Babel and outperformed it because the scope is limited to the most recent browser and Node.js. However, it does not offer all necessary functionalities for the application.

A new experimental JavaScript toolchain considered is Rome[45]. Like Babel, Rome seems to offer many useful functionalities such as code parsing and transformation. An interesting fact about Rome.js is that it has the same author as Babel and Yarn[60], and Facebook is the owner. Unfortunately, Rome is a very recent library as of 2020 Q1 and there is not enough documentation to confirm that it could replace Babel yet.

### 4.2.4 AST Explorer

For exploring the [AST](#) concept and the results of a transformation, AST Explorer[8] is a very useful tool. AST Explorer is not meant to be included in the application, but rather to be used as an external tool to help for inspecting [AST](#).

It allows one to write a code fragment and inspect the [AST](#) generated from a selected parser. In the current version 7 of Babel, Babel-Eslint 9 is used. The [Figure 4.2](#) shows an example of how to transform an [AST](#) to manipulate a code fragment. The code fragment is located on the top-left and its generated [AST](#) is on the top-right. Due to the transformation via the [AST](#) on the bottom-left, the name of the variable is now inverted and the transformed code can be seen on the bottom-right. This is only a small code snippet and the result is simple, but it gives an overview of the main steps to transform a code.



Figure 4.2: Screenshot of an abstract syntax tree transformation example taken from AST Explorer.

## 4.3 Architecture

The architecture section presents a high level structure of the system and explains the interactions between the `modules`. Then, the process flow is explained with the support of an activity diagram.

### 4.3.1 Modules

This section details the major parts of the system and the relations between the `modules`. As illustrated in Figure 4.3, some `modules` are part of a greater set and are placed inside colored rectangles. React components and Jest tests are decoupled from the rest of the implementation and have no dependencies on other part of the application.

#### React components

In this section, the `modules` main role is to create React components to be rendered for the UI. These components are the only ones visible for the user and some of them are already described in Section 2.4 User interface.

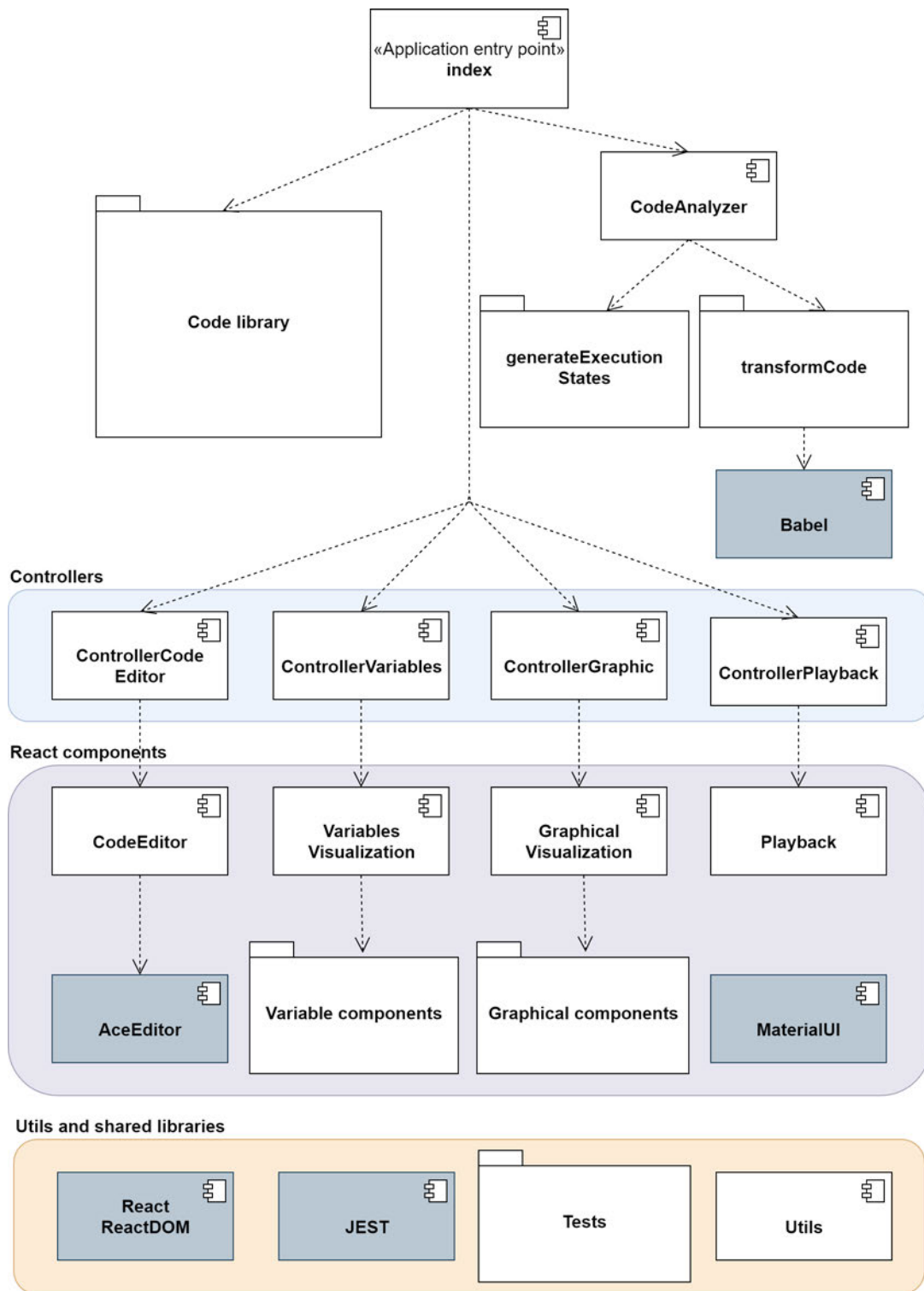


Figure 4.3: Component diagram from a high level structure perspective.

The *VariablesVisualization* module displays a representation of simple variable and their values using the *Variable components* collection. The *GraphicalVisualization* module is responsible to create any graphical visualization type to represent a variable with multiple values such as arrays and lists. It uses multiple components from a collection of *Graphical components* in order to build the graphical visualization.

### Controllers

The controller modules are the interface between the React components and the code analyzer. Their responsibility is to indicate in the [DOM](#) where the components need to be displayed. They also have to manage these components when an event is triggered. The *ControllerCode* module handles and renders the code editor while the *ControllerVariable* and *ControllerGraphic* modules do the same but for the variables. The *ControllerPlayback* module handles keyboard events, the buttons events and renders a slider with the total number of steps.

### Code analyzer

While the previous sections can refer to the body of the application, this part is the brain. The *CodeAnalyzer* controls both *transformCode* and *generateExecutionStates* modules to update and transform the code to generate all execution states for a given code fragment.

The *transformCode* module transforms a code using the [AST](#) via Babel library into an [augmented code](#). The *generateExecutionStates* module provides all functions needed for an [augmented code](#) to generate the executions states. Which means that for each step, all states of the code is saved in a array and make possible to update the value of each variable at a given step as described in [Requirement 3.31](#).

Ensuing [Requirement 2.2](#), additional coding languages could be ultimately implemented in the application. This is the section of the architecture that would need to be adapted for each new programming languages. In order words, the workflow would be similar, but different tools and libraries would be used. Furthermore, the main operation done by the *codeAnalyzer* is asynchronous, thus could take time to execute without blocking the application. Even if not needed with the current solution, the asynchronicity would allow the *codeAnalyzer* to delegate the code processing to a server.

Moreover, the architecture makes the testing of the application easier since the *codeAnalyzer* section is decoupled from the rest. This is the main relevant section to test and it is not dependant on the UI. Hence, it supports the [Requirement 2.5](#) and [2.6](#).

The workflow to transform a code fragment involves six main steps that is illustrated and described in the [Subsection 4.3.2 Process data flow](#). The transformation handles all features listed in [Subsection 2.3.1 Common programming concepts](#).

### **Index, utils and code library**

*Index* is the entry point of the application. It creates instances of various controllers and handles the application main events. It also declares the default code fragment to be used by the application.

The *code library* module refers to the collection of code fragments provided by the application.

The last set of modules called *Utils and shared libraries* refers to some tools that are described in the [Section 4.1 Technologies](#). React, Jest and Material-UI are the main ones. This set also holds the utils custom library that combines all general-purpose functions and methods such as math functions. Lastly, the collection of tests include at least a test for each feature of the coding language as specified in [Requirement 2.5](#).

### **4.3.2 Process data flow**

This section describes the main process flow to transform and analyse a code fragment. The [Figure 4.4](#) shows only the first part of the whole process flow and excludes the last step which is the visual representation. This activity diagram illustrates the process to build a visualization. It starts with a code as an input and ends with a visualization ready to be displayed. If an error exists in the code fragment or the code complexity limit is exceeded as pointed out in [Requirement 3.49](#), the application should notify the user that the build failed because of invalid code. Additionally, the user can modify any selected code fragment or paste his own code at any time. Once a modification occurs in the code editor, the build process is restarted.



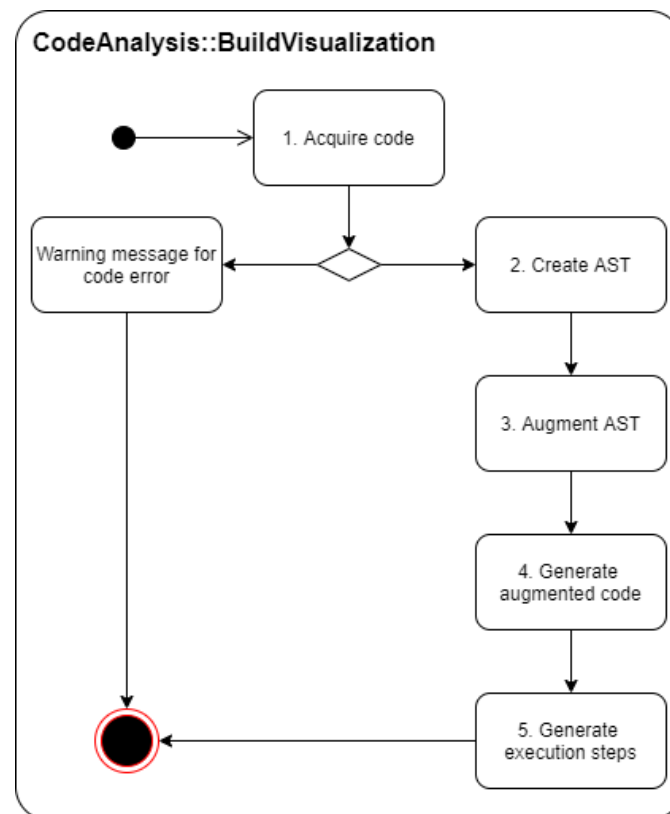


Figure 4.4: Activity diagram of the operational flow for building a visualization.

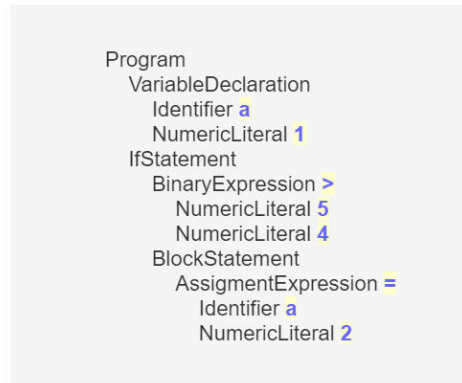
At each step of the process flow, the data is transformed. Figure 4.5 lists the various shapes of the data from the initial code fragment to the final visual representation. The last step represents the visualization of the whole UI, essentially representing the highlight of the code fragment, the current step count of the code execution, the variables and graphical visualization.

### 1. Input code

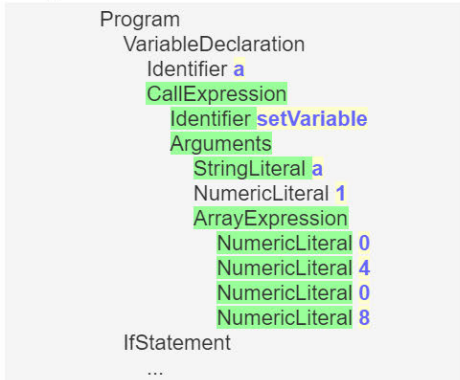
```

1  var a = 1;
2
3  if (5 > 4) {
4      a = 2;
5  }
    
```

### 2. Abstract syntax tree



### 3. Augmented abstract syntax tree



### 4. Augmented code

```

1  var a = setVariable("a", 1,
2      [0, 4, 0, 8]);
3
4  if (evaluateExpression(5 > 4,
5      [2, 4, 2, 8])) {
6      a = setVariable("a", 2,
7          [3, 4, 3, 8]);
8  }
    
```

### 5. Execution steps

```

[ {
  CodeRange: {
    LineStart: 0,
    ColumnStart: 4,
    LineEnd: 0,
    ColumnEnd: 8,
  },
  State: {
    a: 1,
  },
  Target: "a",
}, {
  ...
} ]];
    
```

### 6. Visual representation



Figure 4.5: Data representation at each step of the process flow.

### 1. Acquire code

The process starts when the application gets a code fragment. The code fragment comes from the user or is chosen from the code fragment list provided by the application. This process handles run time errors and the limit of number of lines of code. If an invalid code is entered, the process is blocked until it is rectified.

### 2. Create AST

In the second step, it converts the code fragment into an *AST*, a representation of the code that will allow meta-programming analysis such as extracting the list of variable names. This part is automatically handled by the library Babel and holds a large amount of information for each part of the code.

### 3. Augment AST

Next, the application augments the *AST* with additional intermediate inspection method calls. This new version of the augmented *AST* contains more data like the name of each variable from the code fragment.

### 4. Generate augmented code

This step is the reverse process of creating an *AST*. Instead, the augmented syntax tree is used to generate a new version of the code fragment. This *augmented code* contains all the intermediate inspection methods calls that are intercepting expressions, assignments or any other relevant code constructs. These inspection methods act as a wrapper or a decorator.

### 5. Generate execution steps

Then, the application executes the *augmented code* fragment which should still produce the same result as the initial code fragment. In addition, it generates the *execution steps* data. All states for the *execution steps* are contained in a variable, so the data needed for the meta-programming is available at any step. Each state includes the value of all variables for that execution step and where it corresponds in the code. For example, the meta-programming contains all data needed to highlight part of the code fragment and the visualization and corresponds to the Requirement 3.23, Requirement 3.32 and Requirement 3.37.

### 6. Display visualization

The last step of the process is the visualization of the code fragment execution using the *execution steps* data. The final visualization is composed by the initial code fragment with colored highlights and the output data state.

## 5 Implementation

A successful implementation of this application will result in a dynamic visualization of the memory variables. This means that some data is required at each execution step of the code such as the code range to highlight the current location of the code in the code editor and the state data for each active variable. The state data contains the variable name, its value, scope identifier, parent scope identifier and a boolean variable to indicate if the variable was declared during the current step. Finally, while the `augmented code` is executed, the state data is extracted in order to display the visual representation. However, the initial user-provider code fragment and the calculations produced by the code should remain identical. In other words, the user should not see any modifications or `tracing instructions` added to its code.

This section exposes the details about the code transformation and how the `AST` is augmented. For each step of the process flow listed in the [Figure 4.5](#), a sample of the code implementation is given and a simple code fragment is used to illustrate how the output result of each step looks like.

### 5.1 Parse code

The first step is to acquire a code fragment from the user through a code editor. As an example, a simple code fragment is given and will be used throughout the process.

```
1 const code = "const a = 1;";
```

Listing 5.1: Example of a simple code fragment.

The second step is the creation of the `AST` from this source code. It is completely done by a parser called `Babylon7`, used by the library `Babel`.

In order to use the parser functionality of `Babel`, the library needs to be imported. The function to parse the code is then saved in a variable.

```
1 import {parse} from "@babel/parser";
2 const ast = parse(code);
```

Listing 5.2: Import and function call of babel/parser.

Generally, the AST is not seen by the developer because it is mainly used by other programs for analysis or transformation purposes. As the goal is to transform the tree to add more information, it is important to know the structure and composition of the AST. From the simple code fragment given as an example, the AST created is shown in Figure 5.1. This result can be generated on the online tool AST Explorer[8]. Taking into account that Babel usually contains more information about the program such as scopes, the example is a simplified version of the complete AST.

```
- program: Program {
  type: "Program"
  start: 0
  end: 12
+ loc: {start, end}
  sourceType: "module"
- body: [
  - VariableDeclaration {
    type: "VariableDeclaration"
    start: 0
    end: 12
+ loc: {start, end}
    - declarations: [
      - VariableDeclarator {
        type: "VariableDeclarator"
        start: 6
        end: 11
+ loc: {start, end}
        - id: Identifier = $node {
          type: "Identifier"
          start: 6
          end: 7
+ loc: {start, end, identifierName}
          name: "a"
        }
+ init: NumericLiteral {type, start, end, loc, extra, ...
+1}
      }
    ]
  }
]
```

Figure 5.1: Example of an abstract syntax tree created with the parser Babylon7.

Another view of this AST can be very useful to understand the connections between the nodes. The diagram in Figure 5.2 represents the AST of the simple code with the principal nodes of the program. Since the simple code contains only one line and it is a variable declaration, the *Program* node holds only one direct child node. For multiple lines of code, the tree would grow and include multiple child nodes. The AST is made of simple building blocks that shows the simplicity of its structure. However, as the code gets bigger, the structure becomes rapidly more complex and hard to navigate through.

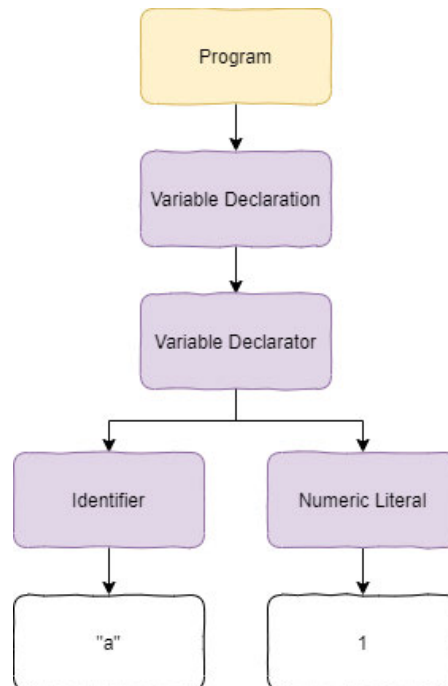


Figure 5.2: Structure of a simple AST generated with Babel.

## 5.2 Augment AST

In the third step of the process flow, the AST is transformed to become the augmented AST. The code to transform the AST have to be carefully developed because it is not given by any tools or library and impact the whole structure of the code. The Babel library offers a way to traverse and transform the tree using the visitor design pattern. The traverse function requires an AST and some methods to be added, modified or deleted. In this case, since the output code must be identical, the functions added

behave as an identity method. The `traverse` function visits each node of the tree, find the ones needed and modify them to include the new methods.

```
1 import traverse from "@babel/traverse";
2 traverse(ast, {
3   VariableDeclarator: transformVariableDeclarator,
4   ForStatement: transformForStatement,
5   IfStatement: transformIfStatement,
6   AssignmentExpression: transformAssignmentExpression,
7 });
```

Listing 5.3: Import and function call of `babel/traverse`.

The `traverse` function contains for now only four methods used for different feature of the code language. Only the `VariableDeclarator` will be used for the example from step one. The `traverse` function will find the `VariableDeclarator` initialization nodes and replace them with the function `transformVariableDeclarator`.

```
1 export const transformVariableDeclarator = (path) => {
2   const variableValue = path.node.init
3     ? path.node.init
4     : identifier("undefined");
5
6   const parentScopeIds = UtilsTransform.getParentScopeIds(path);
7   const parentScopeIdNumericLiterals =
8     parentScopeIds.map(parentScopeId => numericLiteral(parentScopeId));
9   const isDeclaration = true;
10  const variableData = UtilsTransform.getVariableData(
11    path.node.id.name,
12    variableValue,
13    path.scope.uid,
14    parentScopeIdNumericLiterals,
15    isDeclaration,
16  );
17  const codeRange = UtilsTransform.getCodeRange(path);
18  const updateCallbacks = UtilsTransform.getUpdateCallbacks();
19
20  path.node.init = callExpression(
21    identifier("handleVariableDeclarator"),
22    [variableData, codeRange, updateCallbacks]
23  );
24  };
```

Listing 5.4: Code transformation of a variable declaration.

The method takes as a parameter the path where the node was found and prepare the necessary data. The *callExpression* function replaces the node path of the initialization with an identifier called *handle Variable Declarator* that will be explained in the following steps.

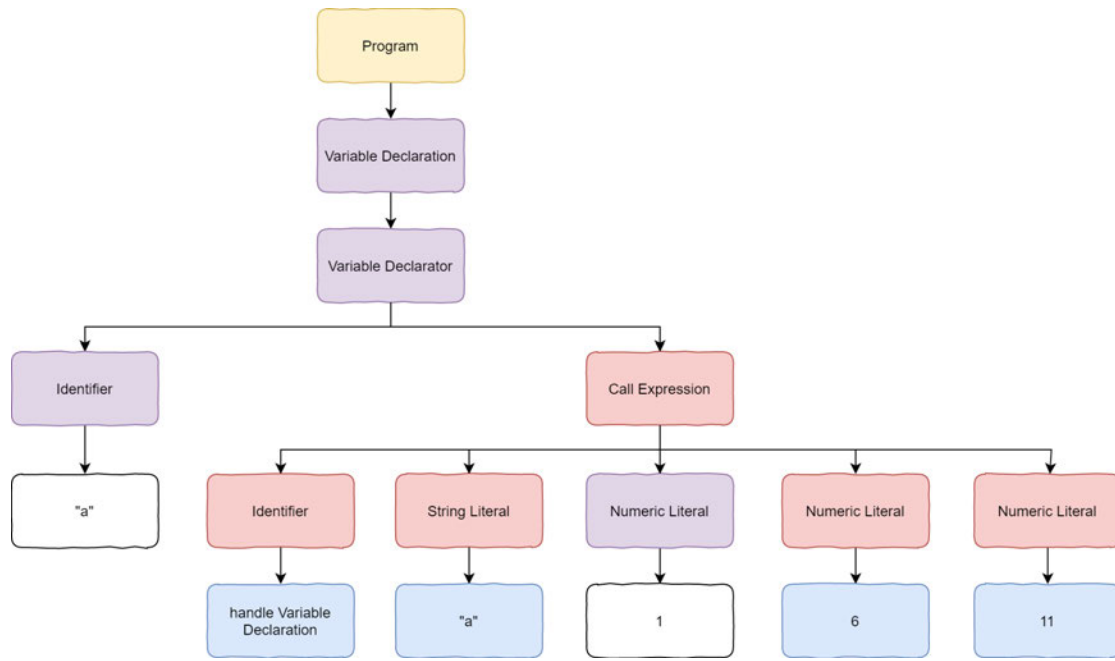


Figure 5.3: Structure of a simple augmented AST generated by Babel.

The Figure 5.3 is a simplified version of the real structure and some nodes are missing such as the scope identifiers. The red color represents the new added nodes and the blue color outline the extra data needed for later steps. The purple *Numeric Literal* node returns the exact value as an expected output of the original code fragment.

It is possible to test an AST transformation with the AST Explorer web application, but since it uses Babel core, the usages of the functions are not the same and require minor modifications. The Appendix D shows how to implement the code using the website to get a similar result.



### 5.3 Generate augmented code

With an augmented AST, an `augmented code` can be built. It is the result of generating the source code from the augmented AST and it can be done using Babel's `generate` function. This step is the inverse operation of parsing.

```
1 import generate from "@babel/generator";
2 const output = generate(ast, code);
```

Listing 5.5: Import and function call of `babel/generator`.

The `augmented code` now contains the `handleVariableDeclaration` function.

```
1 const a = handleVariableDeclarator("a", 1, 6, 11);
```

Listing 5.6: Example of an `augmented code` fragment.

Just like an identity function  $f(x) = x$ , the `handleVariableDeclarator` function will return the same output value as the original source code and all variables keep their initial value.

### 5.4 Generate execution steps

For each step of the code execution, a list of cumulative variables is saved in an array. This array contains information needed to display the visual of the application. In order to create this execution states list, the `augmented code` need to be executed. As new `callExpression` functions were added in the previous step, the execution of the code requires to have access to these handles functions such as the `handleVariableDeclarator`.

The handles functions create a new state which is added to the list of execution states. Finally, as previously mention, it returns the initial output value.

```
1 export const handleVariableDeclarator = (variableData, codeRange, updateCallbacks) => {
2   const {name, value, scopeId, parentScopeIds, isDeclaration} = variableData;
3   const {startIndex, endIndex} = codeRange;
4   const {addExecutionState, updateCumulativeVariables} = updateCallbacks;
5
6   const newValue = Array.isArray(value) ? [...value] : value;
7   const {variables, variableChangeScopeId} = updateCumulativeVariables(name, newValue,
8     scopeId, parentScopeIds, isDeclaration);
9   const variableChange = {name, variableChangeScopeId};
```

```
9
10  const newState = {startIndex, endIndex, variableChange, variables};
11  addExecutionState(newState);
12
13  return value;
14  };
```

Listing 5.7: Code handling of a variable declaration.

This function is sharing all data required to fill the *executionStates* list.

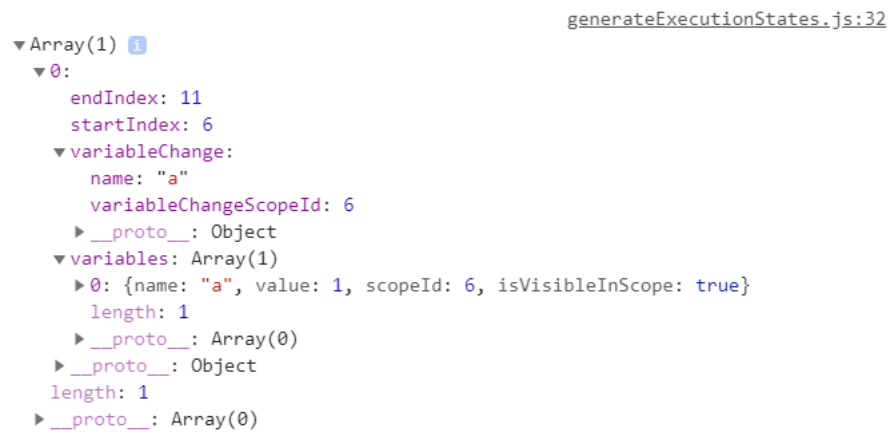


Figure 5.4: Example of an execution steps array.

Figure 5.4 displays an array of only one element because the source code example has indeed only one step. The states contain all data that is needed for the visual representation.

To sum up the process, a string containing a code fragment was transformed with the help of an AST, then executed to construct a list of execution states. These execution states are then used to display the visualization to the user.

## 6 Test

Testing is one of the most important part of building an application because it leads to a better quality assurance. When a feature is built, the first instinct of a developer is usually to run the application and have a quick check if the program works as expected. This testing technique is called *smoke testing* and is very useful, but has its limits. In the absence of proper test strategies, it can lead to a situation where the smoke testing wrongly replaces an exhaustive testing of all features of an application. It is expensive to manually test features during the development of an application and since humans are fallible and imperfect, some features will be forgotten during the process. There are various types of tests, and some of them can even be executed automatically and thus can save development time. Good tests force the development to search for program weakness or prevent system failure. It is also useful to insure the functionalities of the application after modifications.

From unit to integration tests, some examples of tests done for this application are shown and explained. The last part of this section illustrates some testings done on the UI also called End-to-End testing.

### 6.1 Unit tests

Unit tests are useful to test only one function at a time. They are usually cheap to create and test the smallest amount of code possible and are generally numerous in a project. They are used to verify if the code does as expected and behaves properly. These tests should be isolated and not rely on other methods. They give instant feedback, help to reduce debugging time, lower the cost of regression testing and can be run every time a change happens in the code[41]. Jest tool is used for the unit tests of the application.

Multiple unit tests can be done for a single function. As an example, one of the tests for the *clamp* function gives a good idea of a simple unit test.

```

1  test('Value is clamp to the minimum value allowed if the value is smaller.', () => {
2    expect(UtilsMath.clamp(-1, 0, 10)).toBe(0);
3  });

```

Listing 6.1: Code example of a unit test.

All parameters should be at least tested once for the same function. For example, the *clamp* function could receive as parameters a number, a string, an object, an undefined variable, a null variable or a Boolean. This way, almost all possible outcomes can be handled.

```

PASS  src/utils/UtilsMath.test.js (6.5s)
  Utils Math tests
    ✓ Value is clamp to its value if does not exceed the maximum allowed or smaller than the minimum. (7ms)
    ✓ Value is clamp to the minimum value allowed if the value is smaller. (1ms)
    ✓ Value is clamp to the maximum value allowed if the value is greater. (1ms)
    ✓ Value is not clamp and return not a number if it is undefined.
    ✓ Value is not clamp and return not a number if it is an array. (1ms)

```

Figure 6.1: Result of multiple unit tests for clamp function.

Jest is useful to have a quick overview of all tests ran and which ones have failed. It offers a watch mode that can constantly monitor the code and an option to get the code coverage of the entire program. It can be useful to see if a function has not yet been tested. Finally, the overview shows the total time needed to execute all tests. The summary illustrated in Figure 6.2 also includes integrated and puppeteer tests.

```

Test Suites: 2 failed, 8 passed, 10 total
Tests:       6 failed, 60 passed, 66 total
Snapshots:  0 total
Time:        46.659s
Ran all test suites.
npm ERR! Test failed.  See above for more details.

```

Figure 6.2: Summary of all Jest tests.

## 6.2 Integration tests

Integration tests cover a larger part of the application than just a single function. They usually test bundles of functions or subsystems that are depending on each other to

properly work. Moreover, it can require the entire application to be up and running in order to execute these tests.

In this application, the integration tests take a major place. All transform and handler functions required for the code analysis are tested by integration tests. As an example, a simple source code can be tested by an integration test, but rely on other functions like *transformCodeAndGenerateExecutionStates*.

```

1  test("Variable declaration and assignation with const statement produces execution
    states with correct highlight range.", () => {
2    const initialCode = "const a = 1;";
3    const executionStates = transformCodeAndGenerateExecutionStates(initialCode);
4
5    expect(executionStates[0]).toMatchObject({startIndex: 6, endIndex: 15});
6  });

```

Listing 6.2: Code example of an integration test.

As presented in Figure 6.3, different parameters are used to test the variable declaration such as the highlight range, the produced execution states and the correct variable data, etc.

```

PASS src/codeAnalysis/tests/variableDeclarator.test.js (8.338s)
Variable declaration tests
  ✓ Variable declaration and assignation with const statement produces execution states with correct highlight range. (113ms)
  ✓ Variable declaration with let statement produces execution states with correct highlight range. (11ms)
  ✓ Variable declaration from a codeFragment example produces execution states with correct highlight range. (13ms)
  ✓ Variable declaration and assignation with const statement produces execution states with correct variables data. (10ms)
  ✓ Variable declaration and assignation with multiple const statement produces execution states with correct variables data. (32ms)
  ✓ Variable declaration and assignation with multiple const statement on a single line produces execution states with correct variables data. (27ms)
  ✓ Variable declaration and assignation when out of scope produces execution states with correct variables visibility. (27ms)

```

Figure 6.3: Result of tests for a variable declaration.

Integration tests are also useful to test all code fragments of the application and verify if the transformed code produces the same output values as the initial code fragment.

```

1  describe("Code Fragment tests", () => {
2    const testTransformedCodeResult = (initialCode, returnValueCode) => {
3      const transformedCode = transformCode(initialCode);

```

```
4
5   const returnValueInitial = eval(initialCode + "\n" + returnValueCode);
6   const returnValueTransformed = evalTransformedCode(transformedCode + "\n" +
7     returnValueCode);
8
9   return {returnValueInitial, returnValueTransformed};
10 };
11 codeFragmentLibrary.forEach(codeFragment => {
12   test("The " + codeFragment.name + " produces correct output values.", () => {
13
14     const {returnValueInitial, returnValueTransformed} =
15       testTransformedCodeResult(codeFragment.code, codeFragment.testReturnValue);
16
17     expect(returnValueInitial).toStrictEqual(returnValueTransformed);
18   });
19 });
20 });
```

Listing 6.3: Integration test code for validating all code fragment outputs.

The Figure 6.4 exposes failed tests for the code fragment *closure* and *object*, which are not yet correctly implemented in the application. Also, it would be interesting to add more tests to validate multiple sets of input values for each code fragment.

```
FAIL src/codeFragments/codeFragment.test.js (8.869s)
Code Fragment tests
  ✓ The Bubble Sort Algorithm produces correct output values. (113ms)
  ✓ The Selection Sort Algorithm produces correct output values. (34ms)
  ✗ The Closure Feature produces correct output values. (27ms)
  ✓ The Array Collection Feature produces correct output values. (8ms)
  ✓ The Collection Set Feature produces correct output values. (9ms)
  ✓ The Comment Feature produces correct output values. (6ms)
  ✓ The For Loop Feature produces correct output values. (8ms)
  ✓ The Function Arrow Feature produces correct output values. (6ms)
  ✓ The Function Declaration Feature produces correct output values. (4ms)
  ✓ The Function Expression Feature produces correct output values. (5ms)
  ✓ The Hiding Feature produces correct output values. (11ms)
  ✓ The If Feature produces correct output values. (7ms)
  ✓ The If Else Feature produces correct output values. (11ms)
  ✓ The If Else & Else Feature produces correct output values. (15ms)
  ✗ The Object Feature produces correct output values. (17ms)
  ✓ The Operator Feature produces correct output values. (19ms)
  ✓ The Primitive Type Feature produces correct output values. (15ms)
  ✓ The Recursion Feature produces correct output values. (8ms)
  ✓ The Scope Feature produces correct output values. (10ms)
  ✓ The While Loop Feature produces correct output values. (4ms)
```

Figure 6.4: Result of tests for all code fragments.

### 6.3 End-to-End tests

End-to-End tests, also called [E2E](#) are recording and playback style tests that run in a web browser. These tests are mimicking actions that a user could do while using the application. They are testing the functionalities of the [HTML](#) and the [DOM](#) interaction. Puppeteer and Jest are used to test the [UI](#) of the application. For all tests to be executed, Puppeteer launches automatically a Chrome browser. The browser opens a page for each test. Once a test start, it is possible to see all the actions predetermined by the test. All functions of these tests work with *promise* object. A *promise* value is not known at its creation and need to eventually complete in order to output a value.<sup>[11]</sup> It is also possible to turn on the headless option to avoid opening the browser.

The next code test represents a simulation of a user trying to click on the playback *Previous* button while the step is already in the first step. Many other tests for the same button are done to test different situations.

```

1 test("Clicking on Previous button once while in the first step does not change the
  current step.", async () => {
2   const stepCounterIndex = await getCurrentStepIndexAsync(page);
3   await clickDomAsync({dataId: "button-playbackControlPrevious"}, page);
4   const stepCounterIndexAfterPrevious = await getCurrentStepIndexAsync(page);
5
6   expect(stepCounterIndexAfterPrevious).toBe(stepCounterIndex);
7 }, timeout);

```

Listing 6.4: Code example of a E2E test.

The Figure 6.5 highlights a summary of the successful and failed tests.

```

FAIL src/test/uiPlaybackControls.test.js (42.326s)
  Test Playback Controls Play button
    ✓ Clicking on Play button and waiting 2 seconds increases the current step by at least 2 steps. (8032ms)
    ✓ Clicking on Play button should toggle the button to a Pause button. (583ms)
  Test Playback Controls Next button
    ✓ Clicking on Next button increases the current step by exactly one step. (6694ms)
    ✓ Clicking on Next button twice increases the current step by exactly two steps. (4642ms)
    ✓ Clicking on Next button once after clicking on LastStep button increases the current step to the last step. (1165ms)
  Test Playback Controls Previous button
    ✗ Clicking on Previous button once while in the first step does not change the current step. (590ms)
    ✓ Clicking on Previous button once while in the third step decreases the current step by exactly one step. (1717ms)
    ✓ Clicking on Previous button after clicking on FirstStep button does not change the current step. (1125ms)
  Test Playback Controls FirstStep button
    ✓ Clicking on FirstStep button once while in the first step does not change the current step. (584ms)
    ✗ Clicking on FirstStep button once while in the third step decreases the current step to the first step. (1689ms)
    ✗ Clicking on FirstStep button once while in the last step decreases the current step to the first step. (1092ms)
    ✗ Clicking on FirstStep button twice while in the last step decreases the current step to the first step. (1624ms)
  Test Playback Controls LastStep button
    ✓ Clicking on LastStep button once while in the first step increases the current step to the last step. (541ms)
    ✓ Clicking on LastStep button once while in the third step increases the current step to the last step. (1614ms)
    ✓ Clicking on LastStep button twice increases the current step to the last step. (1081ms)

```

Figure 6.5: Result of puppeteer tests for the playback controls.

When a test fails, Jest can provide some verbose help to explain the problem. As an example, this test reveals that the step counting starts at 0, when it is expected to start at the index 1.



- Test Playback Controls FirstStep button > Clicking on FirstStep button once while in the last step decreases the current step to the first step.

```
expect(received).toBe(expected) // Object.is equality

Expected: 1
Received: 0

    97 |     const stepCounterIndexAfterFirstStep = await getCurrentStepIndexAsync(page);
    98 |
  >  99 |     expect(stepCounterIndexAfterFirstStep).toBe(1);
      |                                           ^
    100 |   }, timeout);
    101 |
    102 |   test("Clicking on FirstStep button twice while in the last step decreases the
current step to the first step.", async () => {
```

Figure 6.6: A verbose explanation of a failed test from Puppeteer.

## 6.4 Code testability

In order to run valuable and meaningful tests for this application, some refactorisations were required. In fact, the more tightly coupled the code is, the more difficult it is to test[52]. For the application, a file *UtilsTestDom* was created to facilitate the creation of testing. Among other, the common methods to run Puppeteer tests such as *beforeAllAsync* function to open the page and download the DOM content are contained in this file. Consequently, there are fewer lines to write for each test and less code duplication.

Another approach that could have avoided much of the refactorisation is the Test-driven development (TDD), which is a recommended practice of Agile software development. It implies to concentrate effort on the behavior of the system, instead of focusing on the methods. The idea is to write the tests focusing on the expected behavior of the program, and then implementing the method. It requires a different mindset, since the tests are always one step ahead of the development. It helps to quickly detect if a modification of the code causes the test to fail.[40]

Lastly, a linter like Eslint is a very useful tool to avoid spelling and syntax errors. It help to avoid repeatable errors and highlight the weak spot in the code that is prone to error.

## 7 Conclusion

The initial intention was to develop a proof of concept of a tool to help novice to learn coding and programming concepts. The application would mainly serve academic purposes and handle enough features of a coding language to be able to evaluate the feasibility of this project for future development.

### 7.1 Accomplishments

From the user perspective, this tool gives a way to debug a code without the help of another human or a complex debugger. The user can trust the application to read, highlight dynamic part of the code and display information about the code step by step. The user has the control over the execution of the code and can take the necessary time to analyse it. More than 20 code samples are available and the copy-paste feature for the code editor is functional within the supported feature set. Furthermore, the user interface is simple and has a professional look.

On the development side, every step of this project required a lot of reading, learning and thinking. Indeed, the requirements and design sections were carefully based on many researches about software architecture, memory management, functional and asynchronous programming. Also, a lot of researches were done for choosing tools and languages. Then, a deep study of the [AST](#) concepts and compiler process was necessary. Playing with [AST Explorer](#) was really helpful and it was a good tool to understand the JavaScript syntax and [AST](#) structure. At the end, the whole project gave a strong experience for the many popular web tools such as [NPM](#), [Webpack](#), [React](#), [JavaScript](#), [CSS](#), [HTML](#), [Jest](#), [Puppeteer](#), [Babel](#), [ACE editor](#), [Material-UI](#), [Git](#) and more. Not only these technologies were not trivial to learn, they were challenging to combine together. The setup of a project has always been difficult when it is the first time. On the other hand, without those technologies and tools, this prototype would have been unrealistic, especially without [Babel](#).

## 7.2 Future works

All the requirements with *High* and some with *Medium* level of priority were fulfilled, but there are still other requirements remaining. Thus, three main improvements that could be done are suggested here as possible future works.

First, there are still many features to implement. More graphical visualizations such as trees and graphs could be created to complement the bar diagram. The UI could allow the user to add more visualizations and give him the control on which variables are displayed. More algorithms and code fragments could be added to offer a better coverage of the language features.

Second, the user experience could be improved. As an e-learning platform, the application should have an interface that guides the user through each option. It could offer a tutorial to guide the user on how to use the functionalities of the application and learn the programming concepts in a logical order. Also, code warnings could be added to the platform to handle unsupported features, edge cases and error to clearly communicate to the user.

The last major improvement would be to offer to the user the option to save, share and create their own custom contents on the platform. For example, a teacher could create exercises with various types of graphical visualization and share them with their students. The application is meant to teach, but in the hand of the many users, it might reveals its full potential and even other unexpected usages.

In conclusion, the development of this proof of concept for a learning platform on programming is a clear success. It definitively proved that such a tool is feasible and have a great potential. It highlighted the challenges involved, established the ground base for building a complete application and provided a direction for future works.

# Bibliography

- [1] *Youtube - Angular vs React vs Vue [2020 Update]*.  
<https://www.youtube.com/watch?v=1YWYWyX04JI>. – Accessed:  
2020-06-03
- [2] *Ace - About*. <https://ace.c9.io/#nav=about>. – Accessed: 2020-04-14
- [3] *Github - Acorn*. <https://github.com/acornjs/acorn>. – Accessed:  
2020-04-14
- [4] *Algorithm Visualizer - Home*. <https://algorithm-visualizer.org/>. –  
Accessed: 2020-04-14
- [5] *Angular - Home*. <https://angular.io/>. – Accessed: 2020-04-14
- [6] *Tomassetti - Getting started with ANTLR in C#*.  
<https://tomassetti.me/getting-started-with-antlr-in-csharp/>.  
– Accessed: 2020-04-14
- [7] *Ashley Nolan - The Front-End Tooling Survey 2019 - Results*.  
<https://ashleynolan.co.uk/blog/frontend-tooling-survey-2019-results>. – Accessed: 2020-06-03
- [8] *AST Explorer - Transform*. <https://astexplorer.net/>. – Accessed:  
2020-04-14
- [9] *Babel - Documentation*. <https://babeljs.io/docs/en/>. – Accessed:  
2020-04-14
- [10] *Babel - @babel/types*. <https://babeljs.io/docs/en/babel-types>. –  
Accessed: 2020-06-14
- [11] BROWN, Ethan: *Web Development with Node & Express/Leveraging the JavaScript Stack*. O'Reilly Media, 2019. – 51–54 S. – ISBN 9781492053514

- [12] *Medium - Understanding Component-Based Architecture.*  
<https://medium.com/@dan.shapiro1210/understanding-component-based-architecture-3ff48ec0c238>. – Accessed: 2020-06-03
- [13] *Chai Assertion Library - Home.* <https://www.chaijs.com/>. – Accessed: 2020-04-14
- [14] *Clang - a C language family frontend for LLVM.* <http://clang.llvm.org/>. – Accessed: 2020-04-14
- [15] *Codeinwp - Angular vs React vs Vue: Which Framework to Choose in 2020.*  
<https://www.codeinwp.com/blog/angular-vs-vue-vs-react/#part-4-working-with-the-frameworks>. – Accessed: 2020-06-03
- [16] *Github - cppast.* <https://github.com/foonathan/cppast>. – Accessed: 2020-04-14
- [17] *Youtube - Douglas Crockford. Javascript has a good parts.*  
<https://www.youtube.com/watch?v=DogGMNBZZvg>. – Accessed: 2020-04-14
- [18] *DBD - Debugger framework.*  
<https://docs.python.org/3/library/bdb.html>. – Accessed: 2020-04-14
- [19] *Delicious Brains - Vue vs React: Which is the Best JavaScript Framework in 2020?*  
<https://deliciousbrains.com/vue-vs-react-battle-javascript/>. – Accessed: 2020-06-03
- [20] *ESLint - Home.* <https://eslint.org/>. – Accessed: 2020-04-14
- [21] *JavaParser - Home.* <https://javaparser.org/>. – Accessed: 2020-04-14
- [22] *Wikipedia - Comparison of JavaScript-based source code editors.*  
[https://en.wikipedia.org/wiki/Comparison\\_of\\_JavaScript-based\\_source\\_code\\_editors](https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_source_code_editors). – Accessed: 2020-04-14
- [23] *Jest - Home.* <https://jestjs.io/>. – Accessed: 2020-04-14
- [24] *khan Academy - Computing Computer programming.*  
<https://www.khanacademy.org/computing/computer-programming>. – Accessed: 2020-06-09

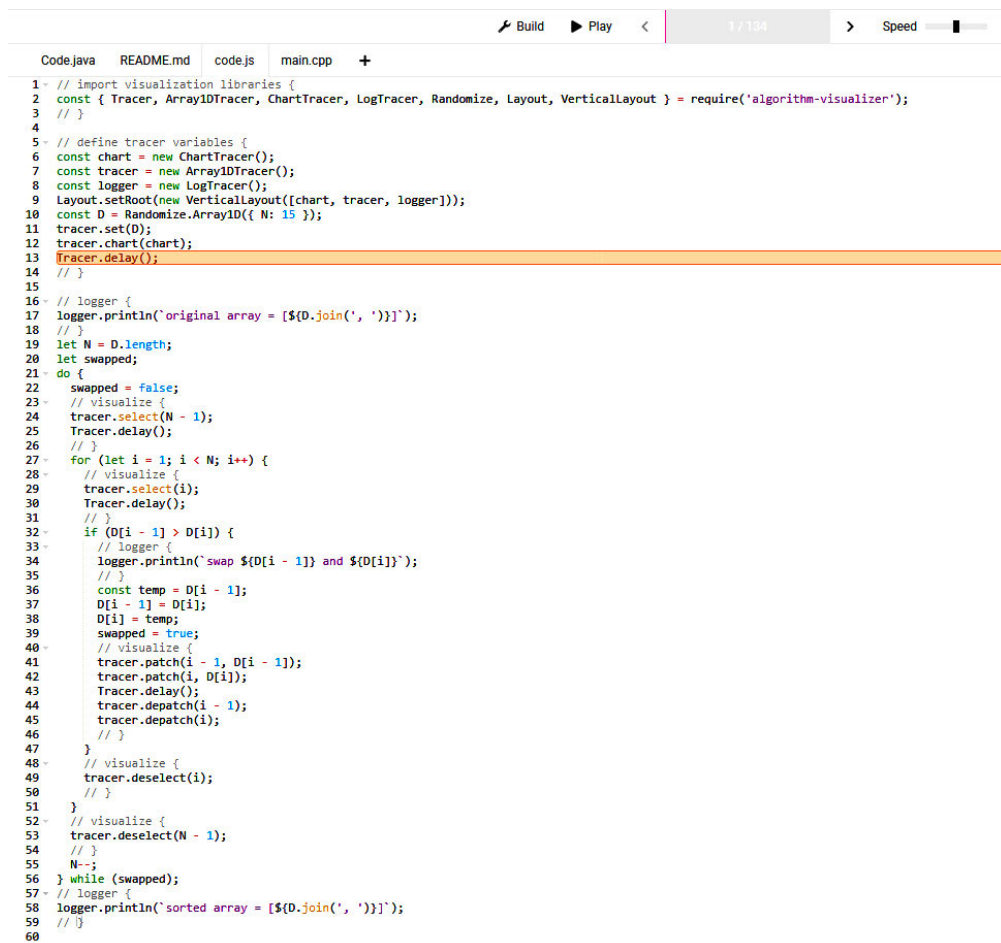
- [25] *Wikipedia - Comparison of programming languages.* [https://en.wikipedia.org/wiki/Comparison\\_of\\_programming\\_languages](https://en.wikipedia.org/wiki/Comparison_of_programming_languages). – Accessed: 2020-04-14
- [26] *Medium - How to Measure Programming Language Complexity.* <https://medium.com/@richardeng/how-to-measure-programming-language-complexity-afe4f7e75786>. – Accessed: 2020-04-14
- [27] *Material Design - Home.* <https://material.io/>. – Accessed: 2020-04-14
- [28] *Material-UI - Home.* <https://material-ui.com/>. – Accessed: 2020-04-14
- [29] *MOCHA - Home.* <https://mochajs.org/>. – Accessed: 2020-04-14
- [30] *Node.js - Home.* <https://nodejs.org/en/>. – Accessed: 2020-04-14
- [31] *Github - NPM.* <https://github.com/isaacs/npm>. – Accessed: 2020-04-14
- [32] *NPM Trends - @angular/core vs react vs vue.* <https://www.npmtrends.com/@angular/core-vs-react-vs-vue>. – Accessed: 2020-06-03
- [33] PARR, Terence: *Language/Implementation/Patterns.* The Pragmatic Programmers, 2010. – 20–36 S. – ISBN 9781934356456
- [34] *Wikipedia - Programming languages used in most popular websites.* [https://en.wikipedia.org/wiki/Programming\\_languages\\_used\\_in\\_most\\_popular\\_websites](https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites). – Accessed: 2020-04-14
- [35] *Github - Puppeteer.* <https://github.com/puppeteer/puppeteer>. – Accessed: 2020-04-14
- [36] *Github - pycparser.* <https://github.com/eliben/pycparser>. – Accessed: 2020-04-14
- [37] *Python - AST.* <https://docs.python.org/3/library/ast.html>. – Accessed: 2020-04-14
- [38] *Python Tutor - Home.* <http://pythontutor.com/>. – Accessed: 2020-04-14
- [39] *Anvil - Running Python in the Web Browser.* <https://anvil.works/blog/python-in-the-browser-talk>. – Accessed: 2020-04-14

- [40] RADY, Ben ; COFFIN, Rod: *Continuous Testing with Ruby, Rails and Javascript*. The Pragmatic Programmers, 2011. – 18–23 S. – ISBN 9781934356708
- [41] RASMUSSEN, Jonathan: *The Agile Samurai: How Agile Masters Deliver Great Software*. The Pragmatic Programmers, 2010. – 197–206 S. – ISBN 9781934356586
- [42] *React - Home*. <https://reactjs.org/>. – Accessed: 2020-04-14
- [43] *Medium - React vs Angular vs Vue.js — What to choose in 2020? (updated in 2020)*. <https://medium.com/techmagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d>. – Accessed: 2020-06-03
- [44] *RedMonk - The RedMonk Programming Language Rankings: January 2020*. <https://redmonk.com/sograzy/2020/02/28/language-rankings-1-20/>. – Accessed: 2020-04-14
- [45] *Romejs - Home*. <https://romejs.dev/>. – Accessed: 2020-04-14
- [46] *Github - Roslyn*. <https://github.com/dotnet/roslyn>. – Accessed: 2020-04-14
- [47] *Sebastian De Deyne - Why I prefer React over Vue*. <https://sebastiandedeyne.com/why-i-prefer-react-over-vue/>. – Accessed: 2020-06-03
- [48] *Spoon - Source Code Analysis and Transformation for Java*. <http://spoon.gforge.inria.fr/>. – Accessed: 2020-04-14
- [49] *Insights StackOverflow - 2020 Developer Survey*. <https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted>. – Accessed: 2020-06-03
- [50] *Github - Sucrase*. <https://github.com/alingpierce/sucrase>. – Accessed: 2020-04-14
- [51] *Tiobe - TIOBE Index for April 2020*. <https://www.tiobe.com/tiobe-index/>. – Accessed: 2020-04-14
- [52] TROSTLER, Mark E.: *Web Development with Node & Express: Leveraging the JavaScript Stack*. O'Reilly Media, 2013. – 44–45 S. – ISBN 9781449323394
- [53] *V8 - Home*. <https://v8.dev>. – Accessed: 2020-04-14
- [54] *Valgrind - Home*. <https://valgrind.org/>. – Accessed: 2020-04-14

- [55] *VISUALGO - Home*. <https://visualgo.net/en>. – Accessed: 2020-04-14
- [56] *VISUALGO - Brute Force, Bubble Sort*. <https://visualgo.net/en>. – Accessed: 2020-04-14
- [57] *Vue - Home*. <https://vuejs.org/>. – Accessed: 2020-04-14
- [58] *Vue.js - Comparison with Other Frameworks*.  
<https://vuejs.org/v2/guide/comparison.html>. – Accessed: 2020-06-03
- [59] *Webpack - Home*. <https://webpack.js.org/>. – Accessed: 2020-04-14
- [60] *Yarn - Home*. <https://yarnpkg.com/>. – Accessed: 2020-04-14



# A Appendix



```
Code.java  README.md  code.js  main.cpp  +
1 // import visualization libraries {
2 const { Tracer, Array1DTracer, ChartTracer, LogTracer, Randomize, Layout, VerticalLayout } = require('algorithm-visualizer');
3 // }
4
5 // define tracer variables {
6 const chart = new ChartTracer();
7 const tracer = new Array1DTracer();
8 const logger = new LogTracer();
9 Layout.setRoot(new VerticalLayout([chart, tracer, logger]));
10 const D = Randomize.Array1D({ N: 15 });
11 tracer.set(D);
12 tracer.chart(chart);
13 tracer.delay();
14 // }
15
16 // logger {
17 logger.println('original array = ${D.join(', ')}');
18 // }
19 let N = D.length;
20 let swapped;
21 do {
22   swapped = false;
23   // visualize {
24   tracer.select(N - 1);
25   Tracer.delay();
26   // }
27   for (let i = 1; i < N; i++) {
28     // visualize {
29     tracer.select(i);
30     Tracer.delay();
31     // }
32     if (D[i - 1] > D[i]) {
33       // logger {
34       logger.println(' swap ${D[i - 1]} and ${D[i]}');
35       // }
36       const temp = D[i - 1];
37       D[i - 1] = D[i];
38       D[i] = temp;
39       swapped = true;
40       // visualize {
41       tracer.patch(i - 1, D[i - 1]);
42       tracer.patch(i, D[i]);
43       Tracer.delay();
44       tracer.depatch(i - 1);
45       tracer.depatch(i);
46       // }
47     }
48     // visualize {
49     tracer.deselect(i);
50     // }
51   }
52   // visualize {
53   tracer.deselect(N - 1);
54   // }
55   N--;
56 } while (swapped);
57 // logger {
58 logger.println('sorted array = ${D.join(', ')}');
59 // }
60
```

Figure A.1: Example of a cluttered code taken from Algorithm-visualizer[56].

## B Appendix

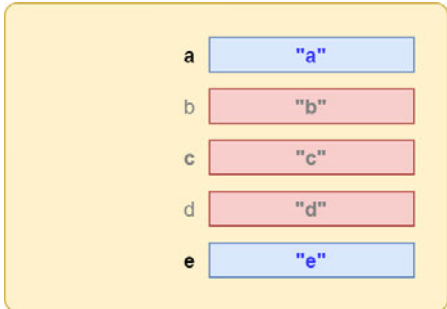
Visualization	Block scope
	<pre>1 // scope 0; 2 const a = "a"; 3 4 { // scope 1, parent 0 5   const b = "b"; 6 7   { // scope 2, parents 1, 0 8     const c = "c"; 9 10    { // scope 3, parents 2, 1, 0 11      const d = "d"; 12    } 13  } 14 } 15 16 { // scope 4, parent 0 17   const e = "e"; 18 }</pre>

Table B.1: Variable visualization for block scopes.

Visualization	Object
	<pre> 1  const obj = { 2    name: "allo", 3    items: [1, 2, 3], 4    props: { 5      age: 32, 6    }, 7    isValid: true, 8    moreProps: { 9      grade: "A", 10   }, 11   identity: x =&gt; x, 12   color: undefined, 13 }; 14 15 const a = 1; 16 obj.moreProps.a = a; 17 obj.moreProps.self = obj; 18 obj.moreProps.other = obj.props; </pre>

Table B.2: Variable visualization for object.

Visualization	Recursion
	<pre> 1  const factorial = (n) =&gt; { 2    if (n &lt; 0) { 3      return; 4    } 5 6    if (n === 0) { 7      return 1; 8    } 9 10   return n * factorial(n - 1); 11 } 12 13 factorial(3); </pre>

Table B.3: Variable visualization for recursion.

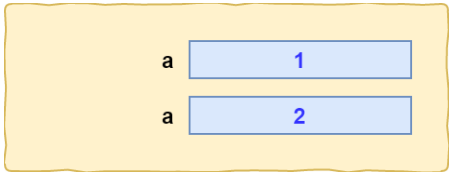
Visualization	Variable hiding
	<pre> 1  const a = 1; 2  console.log(a); // 1 3  if (true) { 4    const a = 2; 5    console.log(a); // 2 6  } 7  console.log(a); // 1 </pre>

Table B.4: Variable visualization for variable hiding.

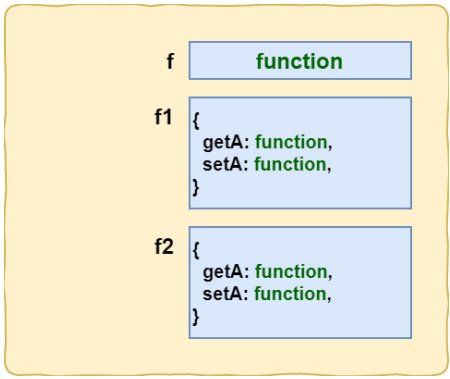
Visualization	Closure
	<pre> 1  const f = () =&gt; { 2    let a = 1; 3    let b = 2; 4 5    return { 6      getA: () =&gt; a, 7      setA: (value) =&gt; a = value, 8    }; 9  } 10 11 const f1 = f(); 12 const f2 = f(); 13 14 f2.setA(3); </pre>

Table B.5: Variable visualization for closure.

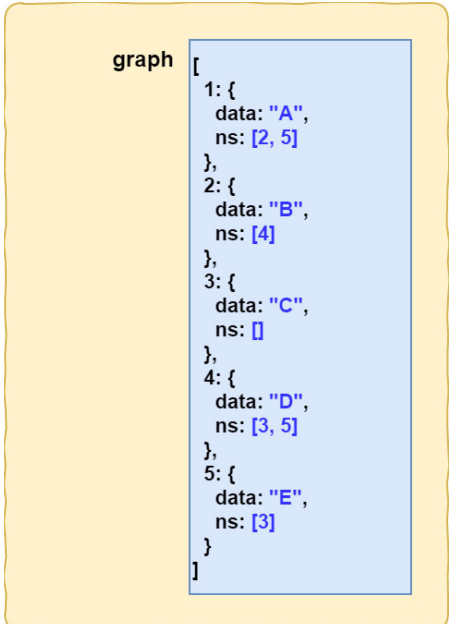
Visualization	Graph
 <pre data-bbox="359 389 667 936">graph [   1: {     data: "A",     ns: [2, 5]   },   2: {     data: "B",     ns: [4]   },   3: {     data: "C",     ns: []   },   4: {     data: "D",     ns: [3, 5]   },   5: {     data: "E",     ns: [3]   } ]</pre>	<pre data-bbox="746 344 1315 591">1 const g = new Map(); 2 3 g.set(1, {data: "A", ns: new Set([2, 5])}); 4 g.set(2, {data: "B", ns: new Set([4])}); 5 g.set(3, {data: "C", ns: new Set()}); 6 g.set(4, {data: "D", ns: new Set([3, 5])}); 7 g.set(5, {data: "E", ns: new Set([3])});</pre>

Table B.6: Variable visualization for graph.

# C Appendix

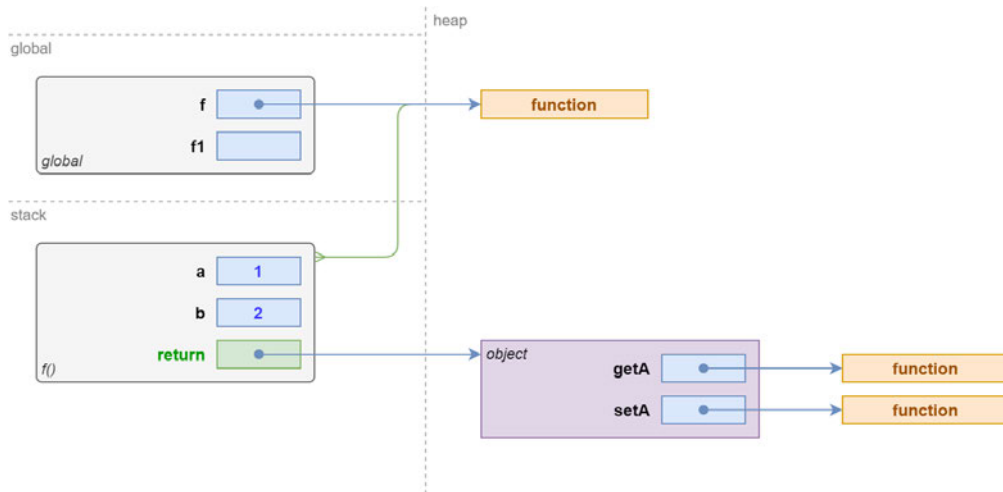


Figure C.1: Advanced memory visualization of a function call that will generate a closure.

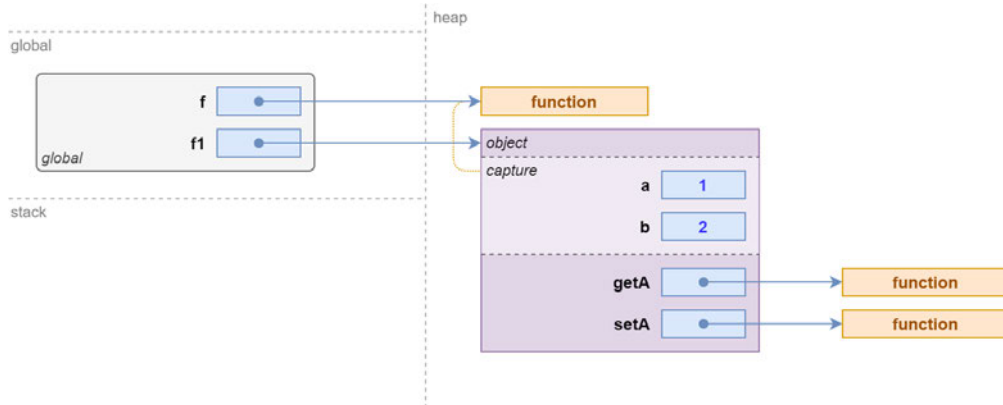


Figure C.2: Advanced memory visualization of a closure.

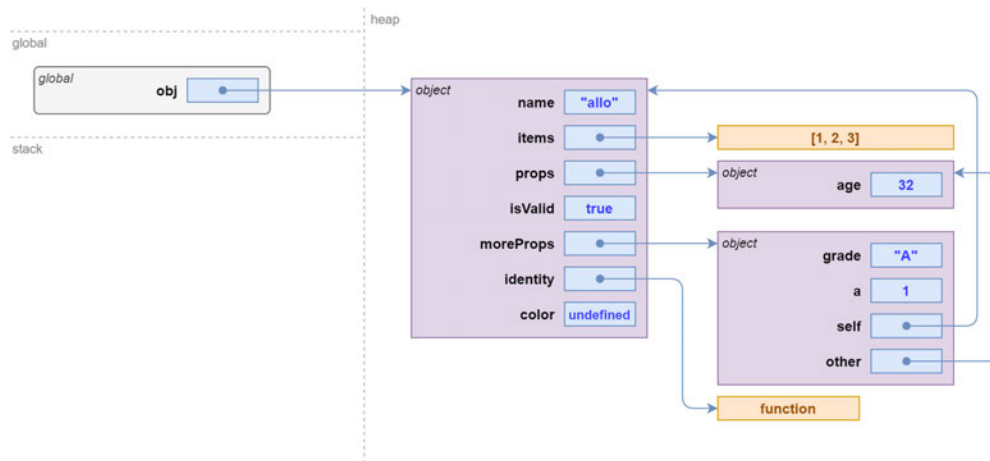


Figure C.3: Advanced memory visualization of an object with values and references.

## D Appendix

```
1 export default function (babel) {
2   const { types: t } = babel;
3
4   return {
5     name: "ast-transform", // not required
6     visitor: {
7       VariableDeclarator: function(path) {
8         const startIndexNode = babel.types.numericLiteral(path.node.start);
9         const endIndexNode = babel.types.numericLiteral(path.node.end);
10
11         if(path.node.init){
12           const callExpression =
13             t.callExpression(
14               t.identifier('handleVariableDeclarator'),
15               [babel.types.stringLiteral(path.node.id.name), path.node.init,
16                 startIndexNode, endIndexNode]
17             );
18           path.node.init = callExpression;
19         } else {
20           const callExpression =
21             t.callExpression(
22               t.identifier('handleVariableDeclarator'),
23               [babel.types.stringLiteral(path.node.id.name), babel.types.identifier(
24                 "undefined"), startIndexNode, endIndexNode]
25             );
26           path.node.init = callExpression;
27         }
28       }
29     }
30   };
31 }
```

Listing D.1: Example of a code to be used in AST Explorer website using babelv7 parser.



## Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

\_\_\_\_\_  
City

\_\_\_\_\_  
Date

\_\_\_\_\_