TÉCNICO LISBOA

Code-Stepping Regular Expressions in the Browser

Luís Alberto Carvalho de Almeida

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

November 2021

Acknowledgments

I want to thank professor José for accepting me in this thesis and for supporting me along the journey; my friends who provided me with support in the last few months; my girlfriend who was my cornerstone during this whole process; and lastly my family, who always pushed me to become a better version of myself.

Resumo

Apresentamos o REXSTEPPER, um *debugger* de referência de expressões regulares JavaScript no browser. O REXSTEPPER foi implementado sobre o REXREF, uma implementação de referência de expressões regulares JavaScript (ECMAScript 5), que transpila uma expressão regular para uma função de JavaScript que reconhece expansões da expressão regular em causa. O REXSTEPPER permite dois modos de visualização: o modo *code-step* e o modo de visualização em árvore. No modo *code-step*, são exibidos individualmente os estados do processo de *matching* e para cada um deles são apresentados atributos que os caracterizam, enquanto no modo de visualização em árvore, os utilizadores conseguem ter uma visão mais global do processo de *matching*, a partir de uma perspetiva em árvore onde cada nó representa um estado do processo. Demonstramos a eficácia do REXSTEPPER utilizando-o para depurar com sucesso um *benchmark* composto por 18 expressões regulares com erros, provenientes dos *websites Stack Overflow* e *Stack Exchange*. O REXREF provou ser uma implementação de referência confiável ao passar com sucesso a todos os testes aplicáveis da suite de testes oficial do JavaScript, Test262.

Palavras-chave: Expressões Regulares, Debuggers, JavaScript

Abstract

We present REXSTEPPER, a reference debugger for troubleshooting JavaScript regular expressions in the browser. REXSTEPPER is implemented on top of REXREF, a trusted reference implementation of JavaScript (ECMAScript 5) regular expressions, which works by transpiling the given regular expression to a JavaScript function that recognises its expansions. REXSTEPPER offers two main visualizations modes: code-stepping mode and tree visualisation mode. In code-stepping mode, users are shown a matching state at a time, showcasing the runtime values that are handled during the matching process, while in tree visualisation mode, users are presented with a global view of the entire matching process as a matching tree, where each node corresponds to a matching state. We demonstrate the effectiveness of REXSTEPPER by successfully using it to troubleshoot a benchmark of 18 faulty regular expressions obtained from the Stack Overflow and Stack Exchange websites. REXREF is shown to be trustworthy, passing all the applicable tests of TEST262, the official JavaScript test suite.

Keywords: Regular expressions, Debuggers, JavaScript

Contents

List of Tables

List of Figures

Chapter 1

Introduction

JavaScript (JS) is the most widespread dynamic language.[1] It is the de facto language for client-side web applications; it is used for server-side scripting with NodeJS and it even runs on small embedded devices [1]. JS regular expressions are an essential part of the language. They are often used for parsing user input and messages coming from the network and they play a key role in securing Web applications as JS developers commonly rely on regular expressions to write complex string sanitisers to prevent a variety of injection attacks [2, 3].

Despite their widespread use, writing regular expressions is a difficult, error-prone task, often leading developers to resort to inefficient trial-and-error approaches. Furthermore, the debugging facilities provided by all browsers and IDEs for JS programming are lacking when it comes to regular expressions. For instance, browsers do not allow programmers to code-step the process of matching a string against a regular expression (onward: regular expression match). Instead, this process is presented to the programmer as a single atomic operation, shedding no light as to why an expected match did not occur, or, alternatively, an unexpected match did occur.

While there has been some research work on the problem of synthesising regular expressions from examples [4–8], much less attention has been given to the development of effective debugging tools for regular expressions. While we are not aware of any such research tool with support for code-stepping regular expression matches, various tools have been proposed with the goal of helping developers visualise their regular expressions [9–11]. These tools construct a visual representation of the given regular expression to help the developer distinguish its key elements and more easily identify bugs. However, these visualisation tools offer no insight into the runtime values that are handled during the matching process (e.g. the value of capture groups). For that, one needs to code-step regular expression matches.

Surprisingly, even outside academia, to the best of our knowledge, not many tools support code-stepping of regular expression matches. We have carefully analysed 26 regular expression debugging tools and discovered that only three come with code-stepping capabilities: Regex 101 [12], Regex Buddy [13], and Regex Coach [14]. When it comes to the debugging of JS regular expressions,

[1]See `https://w3techs.com/technologies/details/cp-javascript`.

all three tools have one major drawback: *they work on stand-alone regular expressions, ignoring their potential interactions with the enclosing programs*. Hence, in order to debug a given regular expression, developers must first use a standard JS debugger, e.g. the browser console, to obtain both the buggy regular expression and the bug-triggering input string, and only then can they use the chosen regular expression debugging tool.

In this thesis, we present RexStepper, the first regular expression debugging tool that allows for code-stepping JS regular expressions without taking them out of their enclosing JS programs. In order to do this, RexStepper must be able to track both the implicit and explicit interactions between regular expression matches and the states of the enclosing JS programs. This is a difficult task as the semantics of JS regular expressions is heavily intertwined with the other aspects of the JS semantics. For instance, several built-in functions of the standard make use of the main regular expression matching function, `RegExp.prototype.exec`, as a subroutine.

RexStepper was designed for debugging regular expressions that occur inside client-side JS programs that run in the browser. In particular, the original program is instrumented so that it records the matching process, generating a matching trace, which can then be inspected using RexStepper when the execution terminates. At debugging time, the developer is allowed to traverse the matching trace to pinpoint the source of a given bug. To streamline this process, RexStepper supports the use of regular expression *break points*, which allow the developer to navigate the control directly to the point where they think the bug might have originated. To the best of our knowledge, RexStepper is the only regular expression debugging tool that offers this capability.

Besides allowing the developer to code-step regular expression matches, RexStepper also provides a mechanism for visualising the entire matching process. More concretely, for each regular expression match, RexStepper constructs a matching tree that illustrates the complete matching process, including the points where that process is forced to backtrack due to a matching failure. We believe that the combination of the code-stepping mechanism with the global view of the entire matching process, provided by the matching tree, greatly improves the overall debugging experience.

In order to evaluate RexStepper, we composed a benchmark consisting of 18 buggy regular expressions obtained from the Stack Overflow and Stack Exchange forums. We identify five common classes of bugs and show how these bugs can be easily clarified through the use of RexStepper. In particular, for each buggy expression, we show how to insert a break point that would lead directly to the bug and how to identify the bug from the inspection of the matching state at the break point.

At the core of RexStepper is RexRef, our new reference implementation of JS regular expressions. RexRef implements ECMAScript 5 [15] regular expressions faithfully, including a complete implementation of the `RegExp` built-in object (Section 15.10 of the standard) as well as implementations of all the methods of the `String` built-in object (Section 15.5) that interact with regular expressions (`match`, `replace`, `search`, and `split`). RexRef was thoroughly tested against Test262 [16], the official JS test suite, passing 718 out of a total of 729 tests. Although there are many academic reference implementations of various versions of the ECMAScript standard [17–22], none of them supports regular expressions, making RexRef the first reference implementation of JS regular expressions.

REXREF was implemented directly in JS. Hence, it can be straightforwardly included in any JS program, allowing us to override the native JS built-in regular expression library. While in this project we use REXREF to enable our code-stepper, we believe that it can also be used to enable other types of analysis of JS regular expressions and the programs that use them. For instance, existing symbolic execution tools for JS [23–26] can use REXREF to offer symbolic reasoning over JS regular expressions without having to implement them natively. Instead, they can simply symbolically execute the code of REXREF when matching a given concrete regular expression against a possibly symbolic string. This type of symbolic reasoning could, in turn, be applied to validate string santisers that make use of regular expressions.

1.1 Contributions

In summary, our contributions are: the first regular expression code-stepper that allows for the integrated debugging of JS regular expressions; and a reference implementation of JS regular expressions that is thoroughly tested against the official JS test suite, passing 98.49% of the applicable ES5 regular expression tests.

1.2 Replication Package

The source code of REXREF and REXSTEPPER is available at [28] and [27], respectively, in the form of replication packages. REXSTEPPER's replication package additionally includes a Web interface [29] for interacting with the tool, which comes with the 18 analysed buggy regular expressions.

Chapter 2

Related Work

There is a vast body of research on regular expressions, covering topics as varied as: symbolic execution for regular expressions [24, 25], regular expression synthesis [4–8, 15], visualisation mechanisms [9–11], and user studies [30–39]. In the following, we focus our analysis of the related work on: user studies (Section 2.1), visualisation mechanisms (Section 2.2), static analysers for regular expressions (Section 2.3), and regular expression synthesis (Section 2.4). The reader is referred to [37] for a recent broad-spectrum survey on techniques for ensuring the correctness of regular expressions.

2.1 User Studies

There is a vast number of empirical studies focused on characterising how regular expressions are (mis)used in practice [30–39]. These studies tackle a wide variety of topics regarding the pragmatics of regular expressions, such as: **(1)** How often are regular expressions used by typical programmers? [31] **(2)** What regular expression patterns hinder understandability? [30] **(3)** What type of debugging infrastructure is more effective when it comes to finding errors/understanding regular expressions? [34] **(4)** What type of tools and techniques students employ when having to write regular expressions? [35] and **(5)** How well are regular expressions tested in practice? [32] **(6)** Do regular expression extraction methodologies yield similar results? Are regular expression characteristics similar across programming languages? [33] **(7)** Are readability, maintainability and usability increased when using a regular expression language that is identical to the surrounding programming language? [38] **(8)** What are the most frequently used features of regular expressions and what are they used for? [39]

Although none of these studies target the use of regular expressions in the context of JavaScript applications, we believe their findings to be indicative, reinforcing our view that developers need better debugging tools for writing regular expressions. In the following, we examine the most relevant papers.

How often are regular expressions used by typical programmers? Paper [31] explores the context in which regular expressions are used, its most common features and the similarity between them. To this end, the authors inquired 18 developers and analysed 4000 open source Python projects from github, having extracted nearly 14000 unique regular expression patterns. The authors concluded that 50%

of developers use regular expressions at least once a week and that the most common use cases for regular expressions are locating content within a file or capturing parts of strings.

What regular expression patterns hinder understandability? The authors of [30] aim at identifying coding patterns that hinder the understandability of regular expressions, which the authors refer to as *code smells*. They argue that once code smells are removed, the code becomes cleaner, more understandable and easier to manage. However, in the case of regular expressions, code smells are not clearly defined, as it is hard to tell what makes a given expression difficult to understand. In order to better characterise regular expression smells, the authors conduct an empirical study on 42 pairs of behaviourally equivalent but syntactically different regular expressions, with 180 participants and evaluate how well they understand various regular expression features. This study concluded that the DFA (Deterministic Finite Automaton) size of a regular expression significantly affects its comprehension; and identified smelly and non-smelly regular expression representations. For instance, the regular expression `aa*` is considered to be a code smell, as it can be replaced by the simpler expression `a+`, with the same meaning. Another example is the regular expression `\d`, which can be replaced with `[0-9]`. The definition of code smell is always subjective to each developer's experience regarding regular expressions.

How well are regular expressions tested in practice? Paper [32] focuses on understanding how thoroughly tested regular expressions in the wild are by examining 1225 open source Java projects on GitHub that include test suites covering their regular expressions. In total, these projects make use of over 15000 regular expressions. The study shows that only 17% of the used regular expressions were tested and, among these, 42% were tested using only a single input. The authors further assess whether or not the usage of an automatic regular expression input generator could lead to more comprehensive testing. To this end, they use the REX [40] symbolic analyser to automatically generate inputs for the collected regular expressions, concluding that both generated test inputs and the programmer-supplied inputs achieve similar coverage levels.

Do regular expression extraction methodologies yield similar results? Are regular expression characteristics similar across programming languages? Paper [33] examines the methodologies used to extract regular expression datasets from online code repositories and tries to characterize the use of regular expression features across different programming languages. The authors focus on the three most popular languages on Github, Python, Java and JavaScript, and draw two main conclusions: various regular expression extraction methodologies yield similar results and regular expression characteristics have significant differences across programming languages.

What type of debugging infrastructure is more effective when it comes to finding errors/understanding regular expressions? In [34], the authors compare textual and graphical notations for regular expressions by conducting a randomized controlled trial with 22 participants. The authors conclude that the length of a regular expression and its notation have a strong impact on its readability, while the participant's background had no measurable effect. Furthermore, the study suggests that graphical representations are more effective than the standard textual ones.

What type of tools and techniques students employ when having to write regular expressions? The authors of [35] conduct an empirical study with 29 students to evaluate which tools and strategies are used when writing regular expressions. The participating students were asked to compose regular expressions that pass certain unit tests created for a specific purpose. They had one hour to complete 20 regular expression tasks, using any tool and with access to the Web. The participant's interactions with Web browsers and the ECLIPSE IDE were recorded by a screen capture software. The authors concluded that: visualization of expressions helps participants pass more tests; those who consulted documentation and tutorials were more likely to pass more tests than those who consulted Q&A websites such as Stack Overflow; and participants who tried to compose the expressions first, instead of searching the web for the solutions, are more likely to pass all tests. Results indicate that current IDEs require better support for the development of regular expressions and that there is a clear need for an automatic mechanism for converting regular expressions between different programming languages.

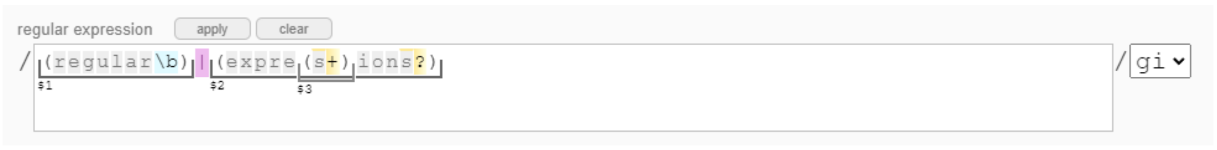
Are readability, maintainability and usability increased when using a regular expression language that is identical to the surrounding programming language? The authors of [38] investigate if a regular expression language which is more similar to the surrounding programming language would improve its overall usability, making regular expressions easier to read and maintain. The authors investigate this using two approaches, namely exploratory interviews and experiments. In the interviews, it was concluded that traditional regular expression syntax is confusing even for experienced developers. As for the experiments, it is shown that having the regular expression language more similar to the surrounding programming language improves the overall usage and development speed.

What are the most frequently used features of regular expressions and what are they used for? Paper [39] studies a dataset consisting of 13K regular expressions extracted from Github Python projects with the goal of finding the most frequently used features of regular expressions in this type of project. For each regular expression in the dataset, the authors produce a *feature vector*, summarising the features of that regular expression. The authors consider a total of 34 features, including the usage of capture groups and the usage of the Kleene operator, `*`. The authors conclude that the most used features are: **(1)** the `+` quantifier; **(2)** capture groups; **(3)** the `*` quantifier; **(4)** custom character classes, such as character ranges; and **(5)** the `.` meta-character.

2.2 Visualisation Mechanisms

Several research projects have tackled the problem of providing visual representations for regular expressions [9–11]. These visual representations are meant to help developers distinguish the key elements of their regular expressions and more easily identify bugs/errors. Visual representation mechanisms can be broadly divided into two main groups: those that completely replace the given regular expression with a new diagram [10, 11] and those that augment the syntax of the given regular expressions with extra visual annotations [9]. REXSTEPPER works both ways. In *code-stepping mode*, REXSTEPPER highlights the part of the regular expression that has already been matched, effectively functioning as a visual augmentation tool. In contrast, in *tree visualisation mode*, REXSTEPPER provides an entirely new diagram (the matching tree) that explains the matching process.

RegViz REGVIZ [9] is a visual augmentation tool for regular expressions. It is a web-based JavaScript tool whose goal is to improve the readability of expressions by presenting the textual and visual representations as one. This is mainly achieved by highlighting and colourizing special tokens, such as quantifiers; structural components, such as logical operators; and other elements, such as assertions and lookaheads, as exemplified in Figure 2.1. The key idea of REGVIZ is to leverage visual cues to reduce the effort required for developers to understand the various components of a regular expression.



```
/(regular\b)|(expre(s+)ions?)/
```

Figure 2.1: Visually augmented regular expression in REGVIZ

SWYN Paper [10] presents SWYN (See What You Need), a tool that helps users constructing regular expressions from examples. This paper was motivated by the lack of visual representation and manipulation capabilities on the inference process of PBE (programming-by-example) systems. In order to tackle this problem, SWYN includes an interface that allows users to observe the effects of the supplied examples on the regular expression being constructed in order to allow the developer to guide the regular expression generation process. SWYN's inference algorithm currently uses an heuristic to incrementally modify the regular expression in response to each new example selected by the user. This process leverages a graph reduction algorithm to identify common elements of the selected examples, as depicted in Figure 2.2. First, it creates a regular expression graph for the first selected example; and then, every new example is added to the graph as a new branch. If there are common elements between the existing graph and that branch, the algorithm merges them. The authors conducted an experiment in which 39 students, who were not familiar with regular expressions, had to perform several regular expression tasks in order to evaluate alternate representations, concluding that graphical notations clearly provide a vast improvement in the usability of regular expressions when users are not familiar with them.

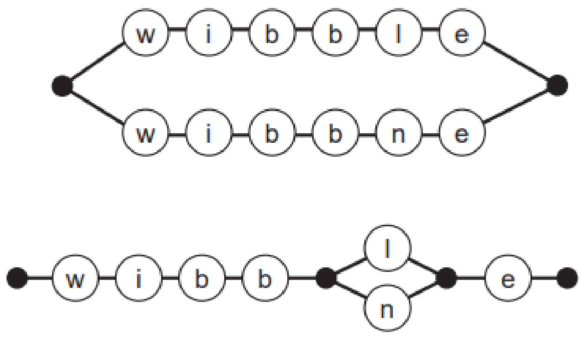


Figure 2.2: Example of SWYN's graph reduction algorithm identyfing common elements between two examples

Visualization and Interaction in automata theory The authors of [11] describe how they incorporated the tools JFLAP [41] and PÂTÉ [42] in their automata theory course to improve the learning experience of their students. The authors claim that students benefit from the interactive visual representations offered by these tools when learning automata, regular expressions and formal grammars.

2.3 Static Analysers for Regular Expressions

To the best or our knowledge, the only academic static analyser for detecting bugs in regular expressions is the tool ACRE (Automatic Checking of Regular Expressions) [43]. ACRE works by performing 11 simple syntactic checks that capture common regular expression bug patterns; for instance: **(1)** mismatched parenthesis, braces and quotes; **(2)** the mis-use of the disjunction character `|`, which has a different meaning depending on whether or not it is used inside a character set; and **(3)** the mis-use of the regular assertions `^` and `$`, which can only be used meaningfully at the beginning and the end of the given regular expression, respectively. The authors evaluate ACRE on a dataset consisting of 826 expressions obtained from the REGEXLIB library [44] and six different Python programs. From the selected regular expressions, 283 were found to be buggy (34%), confirming the need for better tool support for debugging regular expressions in practice.

2.4 Regular Expression Synthesis

In the following, we review several research projects on the topic of (semi-)automatic synthesis of regular expressions. Synthesis algorithms/tools can be broadly divided into two main groups: **(1)** those that create new regular expressions from scratch; and **(2)** those that fix existing regular expressions so that they exhibit the expected behaviour in a given set of inputs. The tools and algorithms in the first group can be further divided into three subgroups, depending on the type of input that they receive; in particular, such tools may receive as input:

- positive and negative examples;
- a natural language description/specification of the behaviour of the regular expression to be synthesised;
- a combination of the first two.

Automate String Processing from Input and Output Examples The authors of [4] performed a thorough study on spreadsheet help forums and concluded that string processing is one of biggest causes of problems for common users in programming. Usually users resort to help forums and ask experts about these problems in the form of input and output examples and it can take several iterations of communication to get the solution. So, the goal of this paper is to introduce a program that replaces these experts and synthesizes user's desired programs only from their examples. In order to tackle this problem, the authors first created a string programming language capable of supporting conditionals, loops and restricted forms of regular expressions; and then created a program synthesis system that can synthesize string processing programs in spreadsheets from input and output examples, written in their language. The synthesizing process is interactive and requires examples to be added at each round in order to improve the generated string processing program, taking an average of one to four rounds of iteration to get to a solution. Regarding performance, the authors claim the algorithm is very efficient, taking only a fraction of a second synthesizing various benchmarks.

How to Fix Regular Expressions in case of Failure? Paper [15] introduces a symbolic approach to fix regular expressions in case of matching failure based on Antimirov partial derivatives [45]. Regular expression partial derivatives represent sets of states of its corresponding NFA and also the sub-terms of the original expression. In case of a failure, partial derivatives allow to trace the failure back to the faulty sub-term in the original expression. As faulty sub-terms are symbols that clash with and input, in order to fix the faulty sub-term, the authors need only to provide an alternative symbol. This novel method is both easy to implement and generally leads to small fixes.

Synthesizing Regular Expressions from examples Paper [5] introduces a method for synthesizing regular expressions from positive and negative examples with the goal of assisting students who are learning regular expressions for the first time. In order to do that, the authors create a fast and interactive synthesizing technique that assures a precise solution, consistent with the given examples. To that end, they created a technique that prunes out the search space effectively while assuring it finds a solution. The key to this algorithm is to over and under-approximate regular expressions to predict whether the current search state can be the final solution or not. This method was implemented in a tool, ALPHAREGEX[46], and was tested against a 25 problem benchmark. The results show that ALPHAREGEX was able to synthesize the desired expressions within an average of 6.7 seconds, which makes it acceptable to be interactively used by students to improve their knowledge of regular expressions.

RFIXER - Repairing incorrect regular expressions from examples Paper [7] introduces RFIXER, a tool for repairing regular expressions from examples. From an incorrect regular expression and sets of positive and negative examples, RFIXER synthesizes the syntactically smallest repair of the original expression that is accepted by the given examples. The main challenge for RFIXER is to find minimal repairs which are scalable to practical regular expressions. The proposed algorithm tackles the scalability problem by leveraging structural properties of regular expressions in order to determine what sub-expressions should be fixed and uses a satisfiability modulo theory solver [47] to efficiently explore the set of possible character classes and numerical quantifiers. The authors evaluated RFIXER by measuring its success rate when trying to produce minimal repairs to regular expressions from three different sources and also by measuring the quality of the given repairs. Regarding effectiveness, they concluded that RFIXER could not solve expressions that require very large fixes or that involve complex nested quantifiers. As for quality, RFIXER produces higher quality repairs than those produced by existing tools and also manages high quality repairs for regular expressions with small alphabets.

Regular Expression Generation from Natural Language and Examples Paper [8] presents a framework for automatically synthesising regular expressions from natural language specifications. The proposed framework works by first using a semantic parser to parse the natural language specification into a sketch, which is basically an incomplete regular expression containing missing components, denoted as holes. Then, this sketch and the positive and negative examples are fed into a program synthesizer, which will instantiate the holes with constructs from their regular expression domain specific language until a consistent regular expression is found. In order to evaluate this framework the authors followed two approaches: **(1)** test it against several English datasets, where the model was able to exploit them, outperforming existing sequence-to-sequence methods, i.e., models trained to convert sequences from a domain into another; **(2)** test it against a dataset of 62 real-world regular expression synthesis problems from Stack Overflow, where it was able to solve 57% of the benchmark, whereas existing deep learning approaches achieve less than 10%.

REGEL - Synthesizing Regular Expressions from Natural Language and examples The authors of [6] propose a tool, named REGEL, which leverages a multi-modal synthesis technique for automatically constructing regular expressions given a combination of natural language descriptions and examples. It first parses the given description into a hierarchical sketch, which is then used by their PBE (programming-by-example) engine to find regular expressions that match the developer's intent. The authors conclude that REGEL achieves an 80% accuracy rate, while a state-of-the-art NLP, DEEPREGEX [48], achieved 43% and a pure PBE approach only achieved 26%. Also, when comparing REGEL's PBE engine with ALPHAREGEX, a state-of-the-art PBE tool, the authors found that REGEL is an order of magnitude faster. In order to further evaluate the effectiveness of REGEL, the authors conducted a user study involving 20 participants where users had to perform 6 regular expression tasks, randomly selected from Stack Overflow and asked the participants to solve half the tasks with REGEL and the other half without it. Results show that REGEL greatly improves the effectiveness of developers: without REGEL users only solved 28.3% of the tasks, while with REGEL they were able to solve 73.3% of the tasks.

Chapter 3

Regular Expression Debugging Tools

In this chapter, we review the existing non-academic tools for code-stepping regular expressions. In order to find such tools, we searched the web for tools and applications for debugging and analysing regular expressions, having obtained a total of 26 tools. Then, we inspected each tool to determine if it came with code-stepping facilities. All inspected tools are listed in Table 3.1 together with their respective URL and a checkmark indicating whether or not they have support for code-stepping regular expressions. Surprisingly, out of the 26 analysed tools, only three do come with code-stepping facilities: REGEX 101 [12], REGEX BUDDY [13], and REGEX COACH [14].

Most tools are very simple and intuitive, having an input box for the regular expression being analysed and another one for the input string to be matched against the given regular expression. Matches are typically represented with colored highlights. Furthermore, some tools make use of additional visualisation cues, aids and/or diagrams to help the user understand the different components/elements of the regular expression being analysed. For instance, DEBUGGEX [49] and CYRILEX [50] represent regular expressions as finite state machines, where each basic expression is mapped to a state and transitions between states represent possible matching paths.

In the following, we analyse the three tools that come with code-stepping facilities, comparing them against each other with respect to the following criteria:

- **Type:** How is the tool distributed to users? Typical options include: web application, desktop application, and/or IDE plugin/extension.
- **Host Language/System:** What is the host language/system of the regular expression language being considered? There are many possible host languages/systems, such as JavaScript, PERL, Python, and the POSIX system. Importantly, some host languages have different regular expression syntaxes.
- **Converts between Flavours:** Is the tool able to transpile a regular expression of a given host language/system to another one, automatically substituting equivalent syntax? For instance, while in JavaScript, the regular expression used to match digits is `\d`, in POSIX it is `[[:digit:]]`.
- **Explanation on Expression / String:** Does the tool provide a textual description of the given

Tool Name	URL	Code-Step
DEBUGGEX	https://www.debuggex.com	✗
REGEX 101	https://regex101.com	✓
REGEXR	https://regexr.com	✗
REGEX BUDDY	https://www.regexbuddy.com	✓
CODVERTER REGEX TESTER	https://codverter.com/src/regextester	✗
KODOS	https://sourceforge.net/projects/kodos/	✗
THE REGEX COACH	http://weitz.de/regex-coach	✓
REXV.2	http://www.rexv.org	✗
REGEX PAL	https://www.regexpal.com	✗
REGULATOR	https://sourceforge.net/projects/regulator	✗
RUBULAR	https://rubular.com	✗
PERL REGEX TUTOR	http://www.perlfect.com/articles/regextutor.shtml	✗
REGEX HERO	http://regexhero.net	✗
CODVERTER	https://codverter.com/src/regextester	✗
NREGEX	http://www.nregex.com/default.aspx	✗
SITE24X7	https://www.site24x7.com/tools/regex-parser.html	✗
CYRILEX	https://extendsclass.com/regex-tester.html	✗
REGEXPLAINED	https://projects.verou.me/regexplained	✗
SCRIPTULAR	https://scriptular.com	✗
VISUAL REGEXP	http://laurent.riesterer.free.fr/regexp	✗
REGEXPER	https://regexper.com	✗
REGULEX	https://jex.im/regulex/#!flags=&re=	✗
LARS OLVA REGEX	http://regex.larsolavtorvik.com	✗
MYREGEXP	http://myregexp.com/	✗
REGEXP TESTER	https://chrome.google.com/webstore/detail/regexp-tester/fekbbmalpajhfifodaakkfeodkpigjbk	✗
REGEX TESTER VISUAL STUDIO	https://marketplace.visualstudio.com/items?itemNam rbangaliyev.RegexTester	✗

Table 3.1: Existing Regular Expression Debugging Tools.

regular expression and of the reasons for the current matching outcome? Typical examples include: having textual aids that describe the matching process and its outcome; for instance: what is the intended behaviour of the basic expressions or flags being used? What was the result of the match? What string(s) were captured?

- **Unit Tests:** Does the tool allow for the creation of positive and/or negative tests/examples? For instance, developers often use a set of examples that the regular expression is supposed to match as a means to validate it.

- **Visual Support:** Does the tool provide any visual cues to improve the user's experience? There are many ways this can be achieved, such as using coloured highlights or tokens to visually separate the components of a regular expression, to map what is being matched in the regular expression with what is being matched in the input string, to indicate what strings were captured in each capture group, etc; or using visual representations of the matching process such as trees or other diagrams.
- **Code-Step:** Does the tool offer the ability to step back and forth on each state of a given match? This facility is a strong advantage when debugging regular expression because users can walk-through the whole matching process one step at a time and clearly observe when the execution starts to show unexpected results.
- **Real-Time Results:** Is the tool able to automatically update its outcome when the given regular expression and/or input string are edited?
- **Free:** Is the tool available for free?

In Tables 3.2 and 3.3 we detail the aforementioned criteria for the three tools that support code-stepping.

Tool Name	Type	Explanation on Expression / String	Unit Tests	Visual Support	Real-Time Results	Converts between Flavors	Free
REGEX 101	Web App	✓	✓	✓	✓	✗	✓
REGEX BUDDY	Desktop App	✓	✗	✓	✓	✓	✗
REGEX COACH	Desktop App	✓	✗	✓	✓	✗	✓

Table 3.2: Existing Regular Expression Debugging Tools With Code-Stepping - Part 1

Tool Name	Host Language/System
REGEX 101	PCRE2, PCRE, ECMA Js, Python, Golang
REGEX BUDDY	AceText 2 and 3, boost::regex, C#, C++Builder, Delphi, EditPad, GNU ERE, Groovy, HTML5, Java, JavaScript, MySQL, Oracle, PCRE, Perl, PHP preg, PostgreSQL, PowerGREP, PowerSheel operators, Python, R, Ruby, Scala, std::regex, Tcl, VBscript, Visual Basic, wxWidgets, XML Schema, XPath, XRegExp
REGEX COACH	PCRE

Table 3.3: Existing Regular Expression Debugging Tools With Code-Stepping - Part 2

3.1 Regex 101

REGEX 101 seems to be the go-to debugger for regular expressions among the community, it is up-to-date, visually appealing and provides a lot of information about regular expressions. It is divided in three main sections: a menu at the left with several components from which we highlight the following: host language selection; unit tests; generation of code; a community library containing regular expressions made by other users; and an account tab that allows users to save custom regular expressions. The middle section contains the main interface, with an input box for the regular expression and another one for the input string(s). This interface is complemented by coloured highlights on both the regular expression and string, over which we can hover the mouse to get more information. Finally, at the right, we have three more components, namely: **(1)** Explanation: a specific section for explaining every basic element of the given regular expression; **(2)** Match Information: a section showing the match result and the captured groups; and **(3)** Quick Reference: a section for quickly searching regular expression elements and their meanings, by category, including practical examples.

As for the code-stepping capability, it only supports it for PERL regular expressions, represented in Figure 3.1. This facility offers the user a clear notion of which step he is in, both in the regular expression and in the input string. Additionally, it represents backtracks with a red arrow pointing backwards. The downsides of this code-stepper are **(1)** it skips whole sub-patterns, for instance, if we match the expression `a+` against the string `aaa`, step 0 is the starting state, matching nothing, and step 1 matches all three a's at a time, instead of just one of them; and **(2)** it does not show the user any runtime values about the match, such as captured groups.

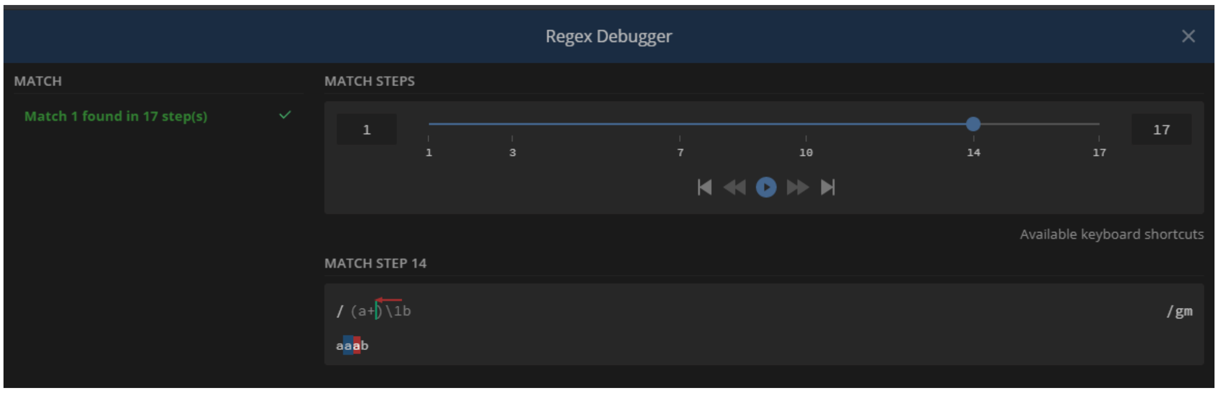
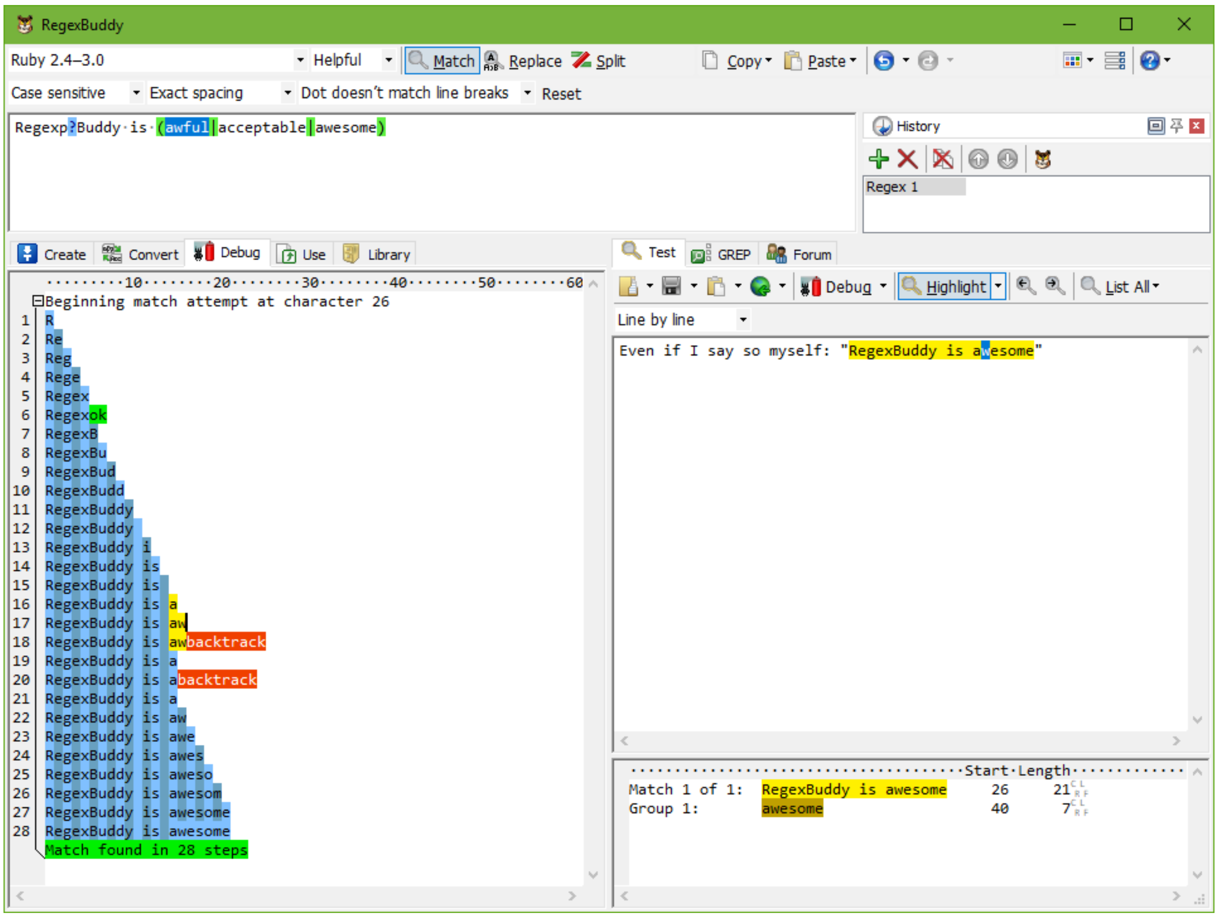


Figure 3.1: Code-stepping in REGEX 101, when matching expression `(a+)\1b` against the string `aaab`

3.2 Regex Buddy

REGEX BUDDY is a paid desktop application, costing 29.95€, it is up-to-date and is one of the most complete tools available, offering an impressive set of approximately 30 regular expression host languages/systems. REGEX BUDDY has a lot of built-in functionalities, such as the Create tab, which explains each basic component of a given regular expression; the Library tab, which includes sample regular expressions from their community; the GREP tab, which enables users to search through a large



Ruby 2.4–3.0 · Helpful
Case sensitive · Exact spacing · Dot doesn't match line breaks

```
Regexp?Buddy·is·(awful|acceptable|awesome)
```

```
·········10········20········30········40········50········60
⊟Beginning match attempt at character 26
1  R
2  Re
3  Reg
4  Rege
5  Regex
6  Regexok
7  RegexB
8  RegexBu
9  RegexBud
10 RegexBudd
11 RegexBuddy
12 RegexBuddy
13 RegexBuddy i
14 RegexBuddy is
15 RegexBuddy is
16 RegexBuddy is a
17 RegexBuddy is aw
18 RegexBuddy is awbacktrack
19 RegexBuddy is a
20 RegexBuddy is abacktrack
21 RegexBuddy is a
22 RegexBuddy is aw
23 RegexBuddy is awe
24 RegexBuddy is awes
25 RegexBuddy is aweso
26 RegexBuddy is awesom
27 RegexBuddy is awesome
28 RegexBuddy is awesome
   Match found in 28 steps
```

Line by line

```
Even if I say so myself: "RegexBuddy is awesome"
```

```
····································Start·Length·············
Match 1 of 1:  RegexBuddy is awesome   26     21
Group 1:       awesome                 40      7
```

Figure 3.2: Code-stepping capability in REGEX BUDDY

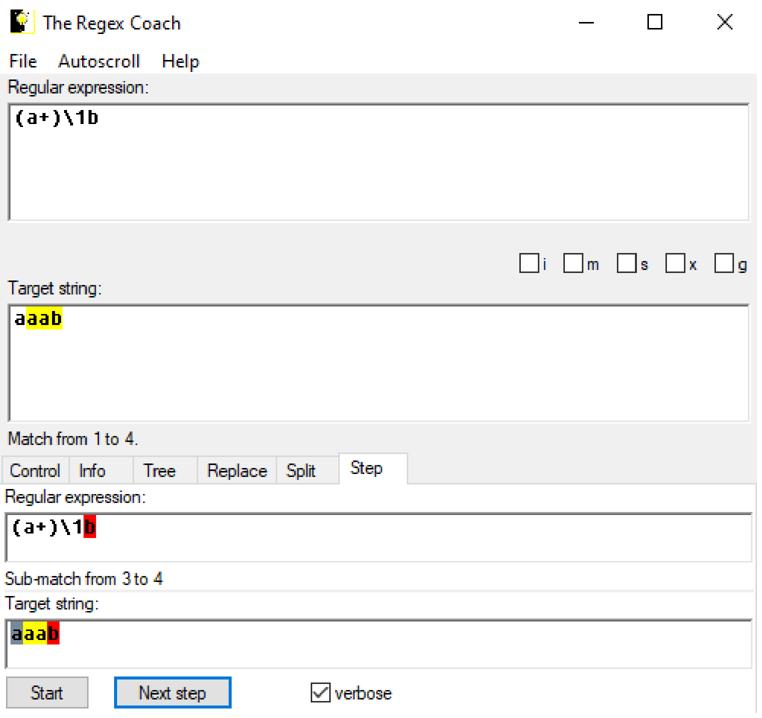
number of files or folders using regular expressions; and the Forum tab, which acts as a help forum where users can ask and answer each other's questions. Additionally, it supports the code-stepping feature (Figure 3.2) by showing a list of all the steps of the matching process over the given input string, including backtrack indicators in red. To the best of our knowledge, REGEX BUDDY is also the only tool that can convert regular expressions from one regular expression host language to another, automatically substituting equivalent syntax. The main downsides of this tool are **(1)** it is not available to users for free; and **(2)** the code-stepping section shows all steps at the same time, instead of letting the user step back and forth through the matching process and analyse them individually, which can be visually overwhelming and make it hard to target a specific state.

3.3 The Regex Coach

REGEX COACH is a free desktop application that despite being very simple is fairly complete. When creating a regular expression, users are offered:

- a textual explanation of what that expression is supposed to match;
- a tree view, allowing users to visualize the composition of regular expression elements;

- a code-stepping feature (Figure 3.3) which allows users to drive the matching process forward one step at a time; however, this tool does not allow users to go back in the matching process, only allowing forward steps;
- a split functionality, where the actual expression is used as the delimiter to split the input string;
- a replace functionality, where the portion of the input string that matches is replaced by a given expression chosen by the user.



Regular expression:

```
(a+)\1b
```

Target string:

```
aaab
```

Match from 1 to 4.

Regular expression:

```
(a+)\1b
```

Sub-match from 3 to 4

Target string:

```
aaab
```

Figure 3.3: Code-stepping in REGEX BUDDY, when matching expression `(a+)\1b` against string `aaab`

3.4 Code-Stepping Limitations

In this section we highlight the main limitations of the three aforementioned tools regarding their code-stepping capability:

- None of the tools supports code-stepping of regular expressions while maintaining their enclosing environments;
- REGEX 101 and REGEX COACH fail to represent which strings were captured by capture groups along the matching process;
- REGEX 101 only allows code-stepping for PERL regular expressions and when matching a quantified sub-expression against a string, it consumes multiple characters in one state;
- REGEX BUDDY's code-stepping feature only details the progress of the matching process with respect to the string and not to the regular expression. Additionally, it does not give the user the ability to navigate through the steps, it simply depicts a list of states in a section of the program.

Chapter 4

RexRef: Reference Implementation of JS Regular Expressions

We describe REXREF, our reference interpreter of JS regular expressions on top of which we developed REXSTEPPER. REXREF is a *reference interpreter* in that it follows the text of the JS standard [15] faithfully and passes all the applicable tests of TEST262, the official JS test suite.

At the core of REXREF is a *regular expression interpreter* that evaluates regular expressions to regular expression *matchers*. A matcher is simply a JS function that recognises expansions of its corresponding regular expression. Besides the regular expression interpreter, REXREF includes: **(1)** a JS transpiler that replaces all occurrences of literal regular expressions in a program with JS expressions that build their corresponding matchers and **(2)** REXREF implementations of all the JS *built-in functions* that interact with regular expressions (e.g. `RegExp.prototype.exec` and `String.prototype.split`).

Before describing the components of REXREF, we give a quick recap of the syntax of JS regular expressions. The syntax of JavaScript regular expressions is given in the table below. It is defined using several auxiliary syntactic classes, of which the most relevant are: characters, $c \in \mathcal{CH}$; boundary assertions, $ba \in \mathcal{BA}$; character ranges, $rg \in \mathcal{CR}$; and quantifiers, $qt \in \mathcal{QT}$.

Syntax of JS Regular Expressions

$$\begin{array}{rl}
c \in \mathcal{CH} & \triangleq \mathcal{CH}_s \cup \mathcal{CH}_m \\
sc \in \mathcal{CH}_s & \triangleq \texttt{ASCII} \cup \texttt{Control} \cup \texttt{Decimal} \cup \texttt{Hex} \cup \texttt{Unicode} \\
mc \in \mathcal{CH}_m & \triangleq \backslash t \mid \backslash n \mid \backslash v \mid \backslash f \mid \backslash r \mid \backslash w \mid \backslash W \mid \backslash d \mid \backslash D \mid \backslash s \mid \backslash S \\
ba \in \mathcal{BA} & ::= \hat{} \mid \$ \mid \backslash b \mid \backslash B \\
rc \in \mathcal{CH}_r & ::= c \mid \backslash b \\
rg \in \mathcal{CR} & ::= rc \mid rc_1 - rc_2 \mid rg_1 rg_2 \\
qt \in \mathcal{QT} & ::= ? \mid * \mid + \mid \{ i \} \mid \{ i, \} \mid \{ i_1, i_2 \} \\
r \in \mathcal{RE} & ::= c \mid ba \mid r_1\, r_2 \mid r_1 \mid r_2 \mid (r)_i \mid (?:r) \mid \backslash i \mid (?=r) \mid (?!\,r) \mid r\, qt \mid r\, qt\, ? \mid [\, rg \,] \mid [\, \hat{}\, rg \,]
\end{array}$$

We explain the syntax of regular expressions bottom-up, first describing the auxiliary syntactic constructs.

Characters, $c \in \mathcal{CH}$, are taken from two distinct sets: the set $\mathcal{CH}_s$ of source characters and the set $\mathcal{CH}_m$ of meta characters. A source character simply denotes itself; it can be written directly in ASCII, or using a decimal, hexadecimal, or unicode escape expression. For instance, `a`, `\97`, `\x61`, and `\u0061` all denote the character `a`. In contrast, meta characters are used to denote pre-established sets of source characters. For example, the meta character $\backslash w$ denotes the set of the lower-case letters, upper-case letters, digits and the underscore character.

Boundary Assertions, $ba \in \mathcal{BA}$, are used to refer to specific points of the given input string, not necessarily related to the characters that occur at those points. The boundary assertions $\hat{}$ and $\$$ respectively denote the beginning and the end of the input string, and the boundary assertions $\backslash b$ and $\backslash B$ respectively denote word boundaries and non-word-boundaries. A word boundary can either be a white space, a line break, or the beginning/end of the input string.

Character Ranges, *rg* $\in \mathcal{CR}$, are used to denote sets of characters in a compact way. A character range is a list of range characters $rc \in \mathcal{CH}_r$ and character intervals $rc_1 - rc_2$. A character range denotes the union of the denotations of the characters and character intervals that it comprises, where the character interval $rc_1 - rc_2$ denotes all of the characters between rc_1 and rc_2, when ordered according to the ASCII codes. For instance, the character range `ad-x` denotes the letter `a` and all the lowercase letters between `d` and `x`.

Quantifiers, $qt \in \mathcal{QT}$, are used to specify the number of times a given regular expression is supposed to be matched. Their meaning is standard: $?$ signifies zero or one time, $*$ signifies zero or more times, $+$ signifies at least one time, $\{\, i \,\}$ signifies precisely i times, $\{\, i, \,\}$ signifies at least i times, and $\{\, i_1, i_2 \,\}$ signifies from i_1 times to i_2 times.

We can now explain the meaning of JavaScript regular expressions, which we divide into the following five groups:

- **Basic Expressions** comprise: *characters*, c, matching the characters denoted by c; *boundary assertions*, ba, checking if ba holds at the current position of the input string; *sequences* of regular expressions, $r_1\, r_2$, matching expansions of r_1 followed by expansions of r_2; and *disjunctions* of regular expressions, $r_1 \,|\, r_2$, matching expansions of either r_1 or r_2. For instance:
  - `^ab|cd$` matches input strings that either start with the characters `ab` or end with the characters `cd`.
  - `a$|cd` matches input strings that either end with character a, or contain the characters cd.
- **Grouping Expressions** comprise: *capturing groups*, (r), matching expansions of r and storing the matching result for future reference; *non-capturing groups*, $(?: r)$, matching expansions of r while not storing the corresponding result; and *backreferences*, $\backslash i$, only matching the most recent match of the i-th labelled capture group. The following expressions exemplify these concepts:
  - `(.*)\1` matches any string consisting of the same string repeated twice; the capturing group $(.*)$

matches any string and stores its value, while the backreference $\backslash 1$ only matches the previously stored value.

  - `(ab)c\1`, identically, the capturing group (ab) matches and stores characters ab, followed by a match of character c and finally the backreference $\backslash 1$ matches what the first group captured, *i.e.* characters ab. An example of a string that would be matched by this expression is $abcab$.

In the following, we assume that each capturing group $(r)_i$ is annotated with a unique index i to be used by our regular expression interpreter. Such indexes can be added to the official regular expression AST via simple pre-processing.

- **Look-Ahead Expressions** comprise: *positive look-ahead expressions*, $(?\!=r)$, matching expansions of r while not updating the matched string; and *negative look-ahead expressions*, $(?!\,r)$, matching strings that do **not** correspond to expansions of r while not updating the matched string. For instance:

  - `x(?=a)(a|b)` matches the string `xa` with matching result `xa`, given that the positive look-ahead is not taken into account for the construction of the match. Analogously, the regular expression `x(?!a)y` matches the string `xy` with matching result `xy` but does not match the string `xay`.
  - `\w+(?=\.com)`, in this regular expression, `\w+` matches one or more alphanumeric characters and the lookahead `(?=\.com)` makes sure that string `.com` exists ahead, but does not take it into account in the final match. For instance, given the input string `google.com`, this regular expression would match `google`;

- **Quantified Expressions** comprise: *greedily quantified expressions*, $r\,qt$, recognising expansions of r the number of times specified by the quantifier qt; and *non-greedily quantified expressions*, $r\,qt\,?$, which stop the matching process once a successful match is found within the bounds of the given quantifier. For instance, when applied to the string `aaa`, the regular expression `a+` produces the matching result `aaa`, while the regular expression `a+?` produces the matching result `a`. More examples include:

  - `a?` matches character a, zero or one time;
  - `a*` matches character a, zero or more times;
  - `a{3,6}` matches character a, three to six consecutive times.

- **Range Expressions** comprise *positive* and *negative* range expressions. A positive range expression, $[$ *rg* $]$, matches any character in the denotation of the range *rg*, while a negative range expression, $[\,\hat{}$ *rg*$]$, matches any character that is not in the denotation of the range *rg*. Examples include:

  - `[abc]` matches either character a, b or c, one time;
  - `[A-Z]` matches any capital letter from A to Z, one time;
  - `[^3t]` matches any character that is not a 3 nor a t, one time.

4.1 Regular Expression Interpreter

Following the JS standard [15], REXREF evaluates regular expressions to regular expression *matchers*. Hence, our regular expression interpreter can be seen as a mathematical function `interp` :: $2^{\mathcal{FL}} \to \mathcal{RE} \to \mathcal{M}$ that given a set of flags *fl* $\subseteq \mathcal{FL}$ and a regular expression $r \in \mathcal{RE}$, produces a matcher $m \in \mathcal{M}$. The set $\mathcal{FL}$ of regular expression flags comprises: **(1)** g for *global*, indicating that the matching process should output all possible matches instead of just the first one, **(2)** i for *ignore case*, indicating that character case is to be ignored when computing matches, and **(3)** m for *multiline*, indicating that new lines should be treated as the beginning of the input string for the purpose of matching boundary assertions.

Interpretation Domains. Regular expression matchers follow the continuation-passing discipline [51, 52] in that each matcher receives as input not only the current matching state but also a continuation representing the matching process to be carried out once the current matcher finishes executing. Formally, a matcher can be thought of as a mathematical function that maps a matching state $\sigma \in \Sigma$ and a continuation $\kappa \in \mathcal{K}$ to a matching outcome $o \in \mathcal{O}$, where: **(1)** a continuation $\kappa \in \mathcal{K}$ is a function that takes a matching state $\sigma \in \Sigma$ and produces an outcome $o \in \mathcal{O}$ and **(2)** an outcome can either be a successful final matching state, denoted by $\mathbf{S}\langle\sigma\rangle$, or a failing final matching state, denoted by $\mathbf{F}\langle\sigma\rangle$. Besides matchers, the regular expression interpreter makes use of the concept of *tester functions*. A tester function $t \in \mathcal{T}$ is simply a function that takes a matching state $\sigma \in \Sigma$ and returns a boolean value indicating whether or not that state satisfies a given regular expression assertion; for instance, if the character at the current index is a non-word character. The table below summarises our regular expression interpretation domains.

Interpretation Domains

MATCHERS	$m \in \mathcal{M}$	::	$\Sigma \to \mathcal{K} \rightharpoonup \mathcal{O}$
TESTERS	$t \in \mathcal{T}$	::	$\Sigma \to \mathcal{B}ool$
CONTINUATIONS	$\kappa \in \mathcal{K}$	::	$\Sigma \to \mathcal{O}$
OUTCOMES	$o \in \mathcal{O}$	::=	$\mathbf{S}\langle\sigma\rangle \mid \mathbf{F}\langle\sigma\rangle$
STATES	$\sigma \in \Sigma$	$\triangleq$	$\mathcal{S}tr \times [\mathcal{S}tr_\perp] \times \mathcal{I}nt$
STRINGS	$s \in \mathcal{S}tr_\perp$	$\triangleq$	$\mathcal{S}tr \uplus \{\text{undefined}\}$
FLAGS	*fl* $\subseteq \mathcal{FL}$	$\triangleq$	$\{g, i, m\}$

Matching States. Matching states, $\sigma \in \Sigma$, bookkeep the matching information during the matching process. More precisely, a matching state σ can be viewed as a triple $(i, \vec{s}, s)$, consisting of: **(1)** the index of the input string to be processed next, i, **(2)** an internal array $\vec{s}$, called *captures array*, mapping capture group indexes to their corresponding captures, and **(3)** the input string on which the outermost matcher was called, s. To better understand how the captures array works, let us consider the two examples given in Figure 4.1.

In the left-hand-side example, we match the regular expression `/(a+)(b*)(c*)/` against the string $\underline{\texttt{a}}\texttt{a}\underline{\texttt{b}}\texttt{bbb}\underline{\texttt{c}}\texttt{cc}$ (ignore underlined characters for now). This expression has three capture groups, each

RE	/(a+)(b*)(c*)/
Str	aabbbbccc
INDEX	CAPTURES
0	[$\perp$, $\perp$, $\perp$]
2	[aa, $\perp$, $\perp$]
6	[aa, bbbb, $\perp$]
FINAL	[aa, bbbb, ccc]

RE	/(a(b*))+(c*)/
Str	ababbbaccc
INDEX	CAPTURES
0	[$\perp$, $\perp$, $\perp$]
2	[ab, b, $\perp$]
6	[abbb, bbb, $\perp$]
FINAL	[a, ϵ, ccc]

Figure 4.1: Matching states and capture groups

corresponding to a different parenthesised subexpression. During the matching process, the matched expansions of the capture groups are stored in the captures array at the corresponding index: i.e., the expansion of the first capture group is stored at the first index, etc. Figure 4.1 shows the content of the captures array before processing the indexes 0, 2, and 6 of the input string, which correspond to its underlined characters, as well as at the final matching state. Initially, all captures are undefined as no expansions of their corresponding capture groups were yet found. In contrast, at index 2, the captures array maps index 0 to the string `aa` as the first capture was already found. The same reasoning applies to index 6 and to the final matching state.

In the right-hand-side example we match the regular expression `/(a(b*))+(c*)/` against the input string ababbbaccc. This example is a bit more involved than the previous one in that this regular expression contains a nested capture group. Capture groups are ordered according to the position of their left-parenthesis in the regular expression source text. Hence, the first capture group corresponds to the expression `(a(b*))`, the second one to `(b*)`, and the third onde to `(c*)`. Importantly, given that the first and second capture groups occur inside a quantified expression, `(a(b*))+`, the values of their corresponding captures are updated every time the enclosing regular expression is matched. For instance, at index 2, the enclosing expression was matched one time, so we have that the first and second capture groups are mapped to ab and b, respectively. At index 6, the enclosing expression was matched two times, so now, the first and second capture groups are mapped to $abbb$ and bbb, respectively. Note, however, that before the matching process completes, the enclosing expression is matched yet a third time. Hence, the values of the first and second capture groups at the final matching state are respectively a and the empty string ϵ, corresponding to the third expansion of the quantified expression.

State Interface. In the implementation, we model matching states as JS objects storing the original input string, the current index, and the captures array. However, to make the interpreter independent of the chosen representation of matching states, we do not interact with the components of matching states directly. Instead, we expose the following methods for accessing/updating their components:

- $\sigma.\mathsf{getS}()$ for obtaining the input string on which the outermost matcher was called;
- $\sigma.\mathsf{getIdx}()$ and $\sigma.\mathsf{setIdx}(i)$ for obtaining and updating the current matching index, respectively;
- $\sigma.\mathsf{getCap}(i)$ and $\sigma.\mathsf{setCap}(i, s)$ for obtaining and updating the i-th capture of σ, respectively;
- $\sigma.\mathsf{getCaps}()$ and $\sigma.\mathsf{setCaps}(caps)$ for obtaining a deep copy of the captures array and updating its value; and
- $\sigma.\mathsf{copy}()$ for creating a copy of the given state.

Matching Combinators. We structure the code of REXREF as a collection of matching combinators, each corresponding to a specific class of regular expressions. In the following, we describe the most relevant combinators. All our combinators are available as part of the REXSTEPPER implementation. To keep the exposition as clear as possible, we present streamlined versions of the presented combinators, focusing on their core functionality and eliding non-instructive technicalities.

CHARMATCHER: A character regular expression c matches the characters denoted by c. Accordingly, the `charMatcher` combinator receives as input a single character `c1` and returns a matcher which checks if the character at the current matching index belongs to the denotation of `c1`. If so, the matcher advances the current matching index by one and calls the given continuation on the current state; otherwise, it returns *failure*. Importantly, a regular expression character may denote a set of string characters. To account for this, we convert the regular expression character to a set of character codes and check if the code of the current character belongs to the computed set.

Character ranges are interpreted similarly, except that one has to compute the set of character codes denoted by the range, which corresponds to the union of the individual ranges included in the character range; for instance:

$$\texttt{codes}(a - zA - Z) = \texttt{codes}(a - z) \cup \texttt{codes}(A - Z)$$

```
function charMatcher (c1) {
        return function(st, cont) {
                var idx = st.getIdx(), c2 = st.getS(idx);
                if (contains(codes(c1), code(c2)) {
                        st.setIdx(idx+1);
                        return cont(st)
                } else { return MakeFail(st) }
        }
}
```

SEQUENCE: A sequence regular expression $r_1\,r_2$ matches expansions of r_1 followed by expansions of r_2. Accordingly, the `seq` combinator receives as input two matchers `m1` and `m2`, respectively corresponding to the first and the second regular expressions, and returns a new matcher that first applies `m1` and then, if it succeeds, `m2`. Given that matchers follow the continuation-passing discipline [51, 52], the generated matcher first creates a new wrapping continuation `cont_d` that captures the computation of `m2` followed by that of `cont` and only then calls `m1` with the argument state and the wrapping continuation `cont_d`.

```
function seq (m1, m2) {
        return function(st, cont) {
                var cont_d = function (st_d) {
                        return m2(st_d, cont)
                }
                return m1(st, cont_d)
        }
}
```

The `seq` combinator trivially lifts to arrays of matchers by applying it recursively and using the *identity matcher* for the base case. The identity matcher simply applies the given continuation to the given state.

```
function mapSeq (ms) {
        if (ms.length == 0)
        return function (st, k) { return k(st) }
        else {
                var m1 = ms.shift(), m2 = mapSeq(ms);
                return seq (m1, m2)
        }
}
```

DISJUNCTION: A disjunction of regular expressions $r_1 \mid r_2$ matches expansions of either r_1 or r_2. Accordingly, the `or` combinator receives as input two matchers `m1` and `m2` and returns a new matcher that first applies `m1` and then, if it fails, applies `m2`. Note that the return matcher succeeds if either `m1` succeeds or `m2` succeeds, only applying `m2` if `m1` fails. Hence, both matchers are called with the given continuation `cont`.

```
function or (m1, m2) {
        return function(st, cont) {
                var r = m1(st, cont);
                if (!isSuccess(r)) {
                        var _st = st.copy();
                        return m2(_st, cont);
                } else { return r }
        }
}
```

GROUP: A group regular expression $(r)_i$ matches expansions of r and stores the matching result in the captures array; note that we annotate group expressions with the index of the corresponding capture group. Accordingly, the `group` combinator receives as input a matcher `m` and a capture group index `i` and returns a new matcher that applies `m` and then saves the substring matched by `m` at i-th position of the captures array. Intuitively, the returned matcher first executes the matcher given as input, then updates the i-th capture group of the resulting matching state, `st_d`, with the substring matched by `m`, and, finally, calls the continuation `cont`.

```
function group (m, i) {
        return function(st, cont) {
                var j1 = st.getIdx();
                var cont_d = function (st_d) {
                        var j2 = st_d.getIdx(),
                                str = st_d.getS(),
                                cap = str.sub(j1, j2);
                        st_d.setCap(i, cap);
                        return cont(st_d)
                }
                return m(st, cont_d)
        }
}
```

BACKREFERENCE: A backreference regular expression $\backslash i$ matches the most recent match of the i-th capture group. Analogously, the `backref` combinator receives an integer `i` and returns a matcher that checks if the string corresponding to the i-th capture group coincides with the next characters to match of the input string. If it does, the matcher moves the current index forward and calls the given continuation `cont` on the input state `st`.

```
function backref (i) {
        return function(st, cont) {
                var s = st.getCap(i), len = s.length,
                        j1 = st.getIdx(), j2 = j1 + len,
                        s_aux = st.getS().sub(j1, j2);
                if (s === s_aux) {
                        st.setIdx(j2);
                        return cont(st)
                } else { return MakeFail(st) }
        }
}
```

ASSERTION: Boundary Assertions, $ba \in \mathcal{BA}$, are used to refer to specific points of the given input string, not necessarily related to the characters that occur at those points; for instance, the boundary assertions $\hat{}$ and $\$$ respectively denote the beginning and the end of the input string. Boundary assertions are evaluated to testers functions, detailed in 4.1.1, which simply take an input state and check if their respective boundary assertions hold at that state. The `assert` combinator is used for lifting a tester function to a matcher function. The returned matcher applies the supplied tester `t` to its state parameter, `st`, to determine whether or not the corresponding assertion holds. If it does, the generated matcher calls the continuation `cont` on the given state and returns its result; otherwise, it returns failure.

```
function assert (t) {
        return function(st, cont) {
                if (t(st)) {
                        return cont(st)
                } else  { return MakeFail(st) }
        }
}
```

LOOK-AHEAD: A positive lookahead regular expression $(?=r)$ matches expansions of r without moving the matching index forward or updating the captures array. Accordingly, the `lookAhead` combinator receives a matcher `m` and returns a new matcher that first applies the matcher `m` and then, if `m` succeeds, resets the matching index and captures array to their original values.

```
function lookAhead (m) {
        return function(st, cont) {
                var caps = st.getCaps(), i = st.getIdx(),
                        r = m(st, cont_id);
                if (isSuccess(r)) {
                        var st_new = r.getState();
                        st_new.setIdx(i);
                        st_new.setCaps(caps);
                        return cont(st_new)
                } else { return r }
        }
}
```

A *negative look-ahead expression*, $(?!r)$, matches strings that do **not** correspond to expansions of r while not updating the matching index and the captures array. Their associated combinator is similar to the one above, so we omit it from the presentation.

REPEAT: A greedily quantified expression $r\,qt$ recognises expansions of r the maximum possible number of times within the bounds of the quantifier qt that will lead to a successful match. As defined in the table below, a quantifier $qt \in \mathcal{QT}$ is evaluated to a pair of integers (i_1, i_2), respectively denoting the upper and lower bounds of the quantifier. For instance, the quantifier $+$ evaluates to the pair $(1, \infty)$, while the quantifier $?$ evaluates to the pair $(0, 1)$.

Quantifier Compilers: $\mathbb{QT} :: \mathcal{QT} \to \mathcal{I}nt \times \mathcal{I}nt$

$$\mathbb{QT}(*) \triangleq (0, \infty) \qquad \mathbb{QT}(+) \triangleq (1, \infty) \qquad \mathbb{QT}(?) \triangleq (0, 1) \qquad \mathbb{QT}(\{n\}) \triangleq (n, n) \qquad \mathbb{QT}(\{n,\}) \triangleq (n, \infty)$$

$$\mathbb{QT}(\{n, m\}) \triangleq (n, m)$$

Greedily quantified expressions are interpreted using the `gRepeat` combinator, which receives as input a matcher `m` together with the minimum and maximum number of times it should be matched, respectively `min` and `max`, and generates a new matcher `m_new` that executes successfully if the matcher `m` can be executed successfully at least `min` times. The new matcher is greedy in that, once the minimum number of matches is reached, it will continue to apply the supplied matcher `m` either until `max` is reached or until it gets a matching failure, in which case it will call `cont` on the matching state corresponding to the last successful match.

```
function gRepeat (m, min, max) {
        return m_new(st, cont) {
                if (max == 0) { return cont(st) }
                var cont_d = function (st_d) {
                        max = max - 1, min = min - 1;
                        return m_new(st_d, cont)
                }
                if (min > 0) { return m(st, cont_d) }
                var old_st = st.copy()
                var ret = m(st, cont_d);
                if (isSuccess(ret)) {
                        return ret
                } else { cont (old_st) }
        }
}
```

We associate *non-greedily-quantified* expressions with their own combinator `ngRepeat`, whose behaviour is analogous to the one described above except that it tries to apply the matcher `m` the minimum possible number of times within the bounds of the quantifier that will lead to a successful match. We omit this combinator as it is analogous to the one given above.

4.1.1 Testers

Boundary assertions $ba \in \mathcal{BA}$ are interpreted as tester functions, which receive an input state and check if their respective boundary assertions hold at that state. In the following, we explain the assertion tester generated for the *word boundary assertion*; the remaining cases are analogous.

The tester generated for the word boundary assertion $\backslash b$ checks if the character at the current processing index, i, is a word character ($s[i] \in \mathcal{WC}$), obtaining the boolean value b_1. Then, the tester checks if the character at the previous processing index, $i-1$, is a word character ($s[i-1] \in \mathcal{WC}$), obtaining the boolean value b_2. Finally, the tester returns b_1 XOR b_2, meaning that it returns `true` if and only if either $s[i-1]$ is a word character and $s[i]$ is not, or vice versa. All assertion testers are defined below:

Beginning of the line assertion `^`

```
function (state) {
        i = state.getIdx();
        if(i == 0) {
                return true;
        }
        else if(isMultiline(flags)) {
                return false;
        }
        else {
                c = state.getS()[i-1]
                return isLineTerminator(c);
        }
}
```

End of the line assertion `$`

```
function (state) {
        i = state.getIdx();
        len = state.getS().length;
        if(i == len - 1) {
                return true;
        }
        else {
                c = state.getS()[i]
                return isLineTerminator(c);
        }
}
```

Boundary assertion `\b`

```
function (state) {
        s = state.getS();
        i = state.getIdx();
        b1 = isWordChar(s[i]);
        b2 = isWordChar(s[i - 1]);
        return (b1 && !b2) || (!b1 && b2);
}
```

Non-Boundary assertion `\B`

```
function (state) {
        s = state.getS();
        i = state.getIdx();
        b1 = isWordChar(s[i]);
        b2 = isWordChar(s[i - 1]);
        return !((b1 && !b2) || (!b1 && b2));
}
```

4.2 Compiling Regular Expressions

RexRef comes with a JS transpiler that replaces all the occurrences of literal regular expressions in a program with the JS expressions that build their corresponding matchers. For instance, the JS program:

```
        var s = /(a*)(b+)\1/.exec("aabaa")
```

is transpiled to the one given in Figure 4.2. We give a stylised version of the compilation for clarity. Observe that the transpiled program creates the matcher corresponding to the original regular expression using the matching combinators discussed in the previous section. Naturally, in order for the transpiled program to run properly, we have to override the native implementation of the `RegExp` constructor with our own implementation, which receives the generated matcher as input.

```
var __matcher1 = mapSeq([
        group(gRepeat(
                charMatcher("a"), 0, Infinity)),
        group(gRepeat(
                charMatcher("b"), 1, Infinity)),
        backref(1)
]);
var s = new RegExp(__matcher1).exec("aabaa")
```

Figure 4.2: Transpiled JS Program

4.3 RexRef Built-in Libraries

REXREF comes with a runtime library that contains JS implementations of all regular expression functions described in the ECMAScript 5 standard (Section 15.5), as well as all string functions that interact with regular expressions (Section 15.0). These JS implementations make use of the matchers generated by our regular expression compiler, and follow their corresponding descriptions in the standard line-by-line. This line-by-line correspondence between the text of a standard and its reference implementations is a well-accepted methodology for establishing trust in reference implementations [17].

We illustrate our approach using the `exec` function (Section 15.10.6.2 of the ES5 standard), whose code is given in Figure 4.3, annotated with the corresponding text of the standard.

Before we go into the details of `exec`, we briefly review how regular expressions are represented in the JS heap. In a nutshell, the evaluation of a regular expression yields a regular expression object. Regular expression objects store the matchers of their corresponding regular expressions in an internal property `[[Match]]`. Furthermore, all regular expression objects share the same prototype, `RegExp.prototype`, which stores all regular expression methods. Our implementation mimics the native one by evaluating regular expressions to regular expression objects and storing the regular expression methods in their shared prototype; the main difference being that we store the matchers in a standard property `__matcher__`, given that ES5 does not allow for direct access to internal object properties.

RegExp.prototype.exec. The `exec` method is supposed to be called on a regular expression object and takes as input the string to be matched against the receiver regular expression. This method recognises the first expansion of the supplied regular expression in the input string and returns an array, storing the matched string at index 0, followed by the bindings of the capture groups at the end of the matching process. Furthermore, the returned array has the additional properties `index`, which stores the index at which the match occurred, and `input`, which stores the input string. Finally, if no match is found, `exec` returns null. Consider, for instance, the execution of `exec` on the regular expression `/(a*)b/g` with input string `cdaadaabcd`; in this case, the returned array object is:

```
      {  0: "aab", 1: "aa",  length: 2, index: 5,  input: "cdaadaabcd" }
```

The core of the `exec` function corresponds to the WHILE loop included in Figure 4.3, which calls the function `match` of the given regular expression at each index of the input string until it either finds a successful match or reaches the end of the input. The function `match`, given in Figure 4.5, is just a

```
// 15.10.6.2 RegExp.prototype.exec(string)
function exec (string) {
        var state_r;
        // 1. Let r be this RegExp object
        var r         = this;
        // 2. Let s be the value of ToString(string)
        var s         = internalToString(string);
        // 3. Let length be the length of s
        var length    = s.length;
        // 4. Let lastIndex be the result of calling [[Get]] internal method of r with argument "last index"
        var lastIndex = r.lastIndex < 0 ? 0 : r.lastIndex;
        // 5. Let i be the value of ToInteger(lastIndex)
        var i         = toInt(lastIndex);
        // 6. Let global be the result of calling [[Get]] internal method of r with argument "global"
        var global    = r.global;
        // 7. If global is false, then i = 0
        if (!global) {
                i = 0;
        }
        // 8. Let matchSucceeded be false
        var matchSucceeded = false;
        // 9. Repeat, while matchSucceeded is false
        while (matchSucceeded === false) {
                // a. If i < 0 or i > length, then
                if (i < 0 || i > length) {
                        // i. Call the [[Put]] internal method of r with arguments: "lastIndex", 0, and true.
                        r.lastIndex = 0;
                        // ii. Return null.
                        return null
                }
                // b. Call the [[Match]] internal method of r with arguments: s and i.
                var ret = r.match(s, i);
                // c. If [[Match]] returned failure, then
                if (isFailure(ret)) {
                        // i. Let i = i+1.
                        i = i+1
                        // d. else
                } else {
                        // i. Let r be the State result of the call to [[Match]].
                        state_r = makeSuccess(ret);
                        // ii. Set matchSucceeded to true.
                        matchSucceeded = true
                }
        }
        // 10. Let e be r's endIndex value
        var e = endIndex(state_r);
        // 11. If global is true
        if (global) {
                // a. Call [[Put]] internal method of r with arguments: "lastIndex", e and true
                r.lastIndex = e;
        }
        // 12. Let n be the length of r's captures array
        var n = nCaps(state_r);
        // 13. Let a be a new array created as if by the expression new Array() where Array is the standard
        ↪  built-in constructor with that name.
        var a = new Array();
        // 14. Let matchIndex be the position of the matched substring withing the complete String s
        var matchIndex = i;
```

Figure 4.3: `RegExp.prototype.exec(string)` - Part 1

```
        // 15. Call the [[DefineOwnProperty]] internal method of a with arguments: "index", Property Descriptor
        ↪  {[[Value]]: matchIndex, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true} and true
        defineOwnProperty(a, 'index', matchIndex, true, true, true);
        // 16. Call the [[DefineOwnProperty]] internal method of a with arguments: "input", Property Descriptor
        ↪  {[[Value]]: s, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true} and true
        defineOwnProperty(a, 'input', s, true, true, true);
        // 17. Call the [[DefineOwnProperty]] internal method of a with arguments: "length", Property Descriptor
        ↪  {[[Value]]: n + 1} and true
        defineOwnProperty(a, 'length', n, false, false, false);

        // 18. Let matchedSubstr be the matched substring (portion of s between i inclusive and e exclusive)
        var matched_substr = s.substring(i, e);
        // 19.  Call the [[DefineOwnProperty]] internal method of a with arguments: "0", Property Descriptor
        ↪  {[[Value]]: matchedSubstr, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true} and true
        defineOwnProperty(a, "0", matched_substr, true, true, true);
        // 20. For each integer i such that i > 0 and i <= n
        var caps = captures(state_r);
        for (var i=1; i<n; i++) {
                // a. Let captureI be the i'th element of r's captures array
                var capture_i = caps[i];
                // b. Call the [[DefineOwnProperty]] internal method of a with arguments: ToString(i), Property
                ↪  Descriptor {[[Value]]: captureI, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]:
                ↪  true} and true
                defineOwnProperty(a, int2str(i), capture_i, true, true, true)
        }
        // 21. Return a
        return a;

}
```

Figure 4.4: `RegExp.prototype.exec(string)` - Part 2

```
function match (str, index) {
        var cont_c = function (st) { return st }
        var st_0 = new State(str, index, this.__caps__);
        return this.__match__(st_0, cont_c);
}
```

Figure 4.5: `__match__` wrapper

wrapper around internal matchers. More concretely, it simply constructs the initial matching state and calls the matcher of the given regular expression with the newly created matching state and the identity continuation.

As a regular expression matcher might be executed multiple times on a given input string (each time starting at a different input index), it is useful, for debugging purposes, to be able to inspect the input index at which the matching process started. In the following, we refer to this index as *global index*.

Chapter 5

RexStepper: Code-Stepping Regular Expressions

REXSTEPPER is, first and foremost, a tool for code-stepping regular expression matches. Hence, it offers a variety of debugging commands for navigating the matching process. Importantly, REXSTEPPER supports the use of regular expression *break points*, which allow the developer to quickly move the control to the point where they think a bug might originate. Before diving into the supported debugging commands we need to clarify the definitions of *iteration* and *execution* in the context of REXSTEPPER. An iteration is the process of matching an expression against an input string, starting from a given index of that string (initially from index 0). In many cases, matching can fail and if that is the case, a new iteration is computed, i.e. the starting index is incremented by one and the matching process starts over from that position instead. REXSTEPPER also supports multiple executions, i.e., a developer can match a given expression against many input strings, for instance, having multiple *exec* calls for the chosen regular expression. Thus, from a hierarchical point of view, executions can contain multiple iterations, which in its turn contains multiple matching states.

Debugging commands include:

(1) *single backward step*, to move the control to the previous matching state;
(2) *single forward step*, to move the control to the next matching state;
(3) *multi backward step*, to move the control to the matching state at the previous break point;
(4) *multi forward step*, to move the control to the matching state at the next break point;
(5) *first step*, to move the control to the starting matching state;
(6) *last step*, to move the control to the final matching state;
(7) *single backward iteration*, to move the control to the previous iteration of the current execution;
(8) *single forward iteration*, to move the control to the next iteration of the current execution;
(9) *single backward execution*, to move the control to the previous execution; and
(10) *single forward execution*, to move the control to the next execution;

When it comes to the implementation of a code-stepper such as REXSTEPPER, there are two complementary strategies. Either one executes the debugger at runtime, effectively interleaving the execution of the program/regular expression being debugged with the execution of the debugger itself, or one runs the debugger only after the execution terminates using information gathered at execution time. As REXSTEPPER is intended to execute in the browser, to follow the first approach, we would have to be able to pause the execution of the running program in order to display a debugging console to the developer and then decide what to do next depending on the developer's input. In the browser, however, user interaction happens mostly asynchronously, with the exception of a few browser commands (e.g. `alert` and `confirm`) that are not fine-grained enough to allow for the implementation of a debugging console. Hence, we opted for the second approach, meaning that, with REXSTEPPER, debugging takes place after the program finishes executing. To this end, we instrument REXREF so that it additionally computes a matching trace containing all the matching states generated during the matching process. REXSTEPPER then parses the generated matching trace and represents it visually, allowing developers to navigate it as they please. In the following, we describe the inner workings of REXSTEPPER, focusing on three main aspects: implementation of break points, runtime instrumentation, and debugging facilities.

5.1 Break Points

In order to cater for the use of break points, we extend the syntax of regular expressions with a distinguished *break point* instruction, •; formally:

$$r^\bullet \in \mathcal{RE}^\bullet ::= r \in \mathcal{RE} \mid \bullet \tag{5.1}$$

In the online tool, we use the sequence of characters `[!]` instead of the symbol • to denote a break point. Break points allow developers to quickly navigate the matching process. For instance, when matching against the regular expression `/(a*) • (b+) • \1/`, the user will first be presented with the matching state after the expansion of `(a*)` is recognised; then, they must decide what to do next. If, for one, they choose to proceed to the next break point, they will be shown the matching state after the expansion of `(b+)` is recognised.

5.2 Runtime Instrumentation

Debugging state interface. To be able to execute debugging commands, we must bookkeep the information generated during the matching process. To this end, we extend the matching state interface with the methods $\sigma.\mathsf{save}()$ and $\sigma.\mathsf{saveBP}()$ to save the current state for later use at debugging time. The main difference between these two methods is that the $\sigma.\mathsf{saveBP}()$ is used specifically to save intermediate matching states associated with break points. Extended states expose various other methods that will be introduced by need.

Bookkeeping matching combinators. We extend the matching combinators introduced in Section 4.1 with the following two combinators, whose goal is to bookkeep intermediate matching states during execution.

```
function saveState (m) {
        return function (st, cont) {
                var cont_d = function(st_d) {
                        st_d.save();  return cont(st_d);
                }
                return m(st, cont_d)
        }
}
function saveStateBP() {
        return function(st, cont) {
                st.saveBP(cont); return cont(st)
        }
}
```

These combinators are simply wrappers around the corresponding extended state methods described above. The first one, `saveState`, returns a new matcher that starts by creating a wrapper continuation that saves the state produced by matcher `m` before calling the supplied continuation, and then calls the given matcher with the wrapper continuation. The second one, `saveStateBP`, is used to bookkeep matching states associated with breakpoints. Hence, instead of calling the method `save` on the state `st`, it calls the method `saveBP`.

```
var __matcher1 = mapSeq([
        group(gRepeat(
                saveState(charMatcher("a"), 0, Infinity))),
        saveStateBP(),
        group(gRepeat(
                saveState(charMatcher("b"), 1, Infinity))),
        saveStateBP(),
        saveState(backref(1))
]);
```

Figure 5.1: Transpiled Regular Expression with Break Point

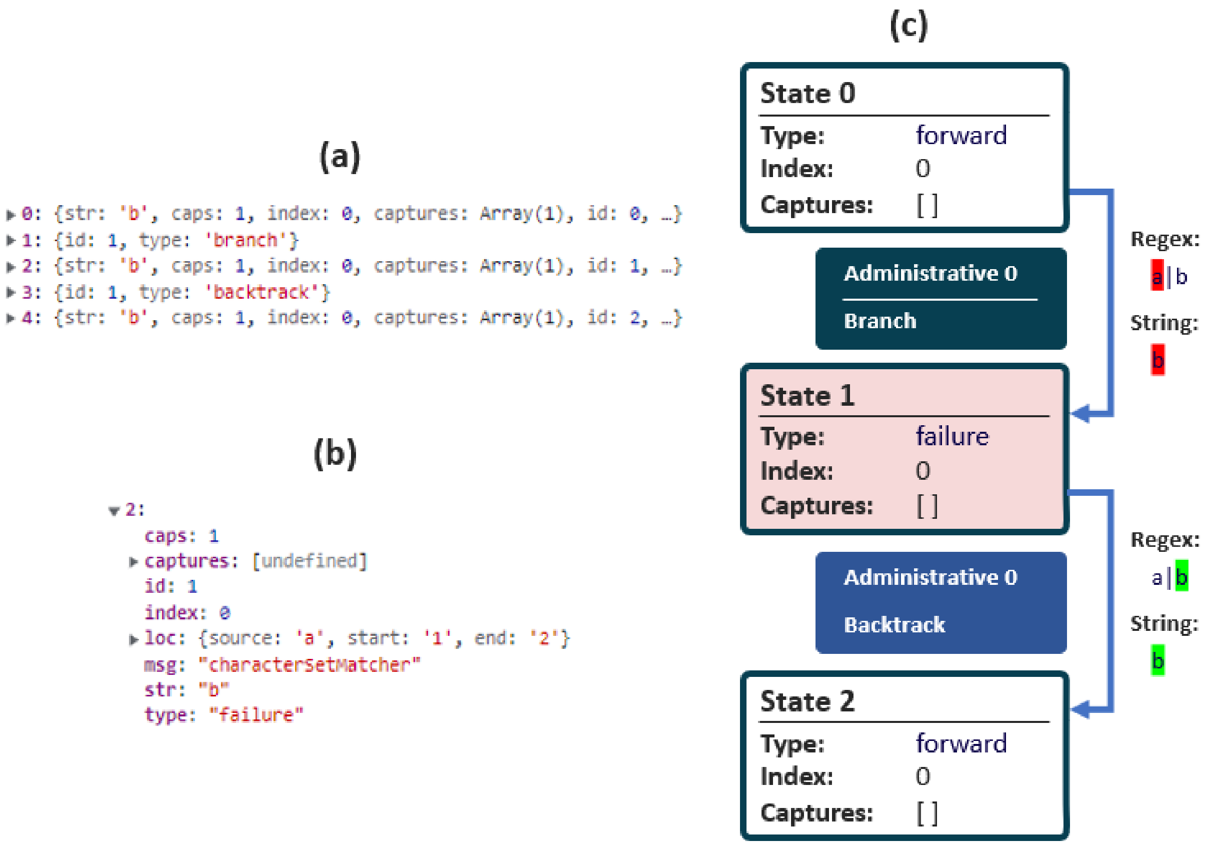
Instrumentation. The RexRef transpiler discussed in Section 4.2 was instrumented to produce the matching trace necessary for RexStepper to work. More concretely, RexRef was modified so as to save the required intermediate matching states. To this end, the generated matchers make use of the combinators `saveState` and `saveStateBP` introduced above. Importantly, we do not have to bookkeep all the states that are generated during the matching process but only those for which the current matching index or the captures array change. Hence, instead of wrapping all intermediate matchers inside a `saveState` combinator, we only wrap the matchers corresponding to: characters, character ranges, backreferences, and capturing groups. For instance, Figure 5.1 shows the stylised compilation of the regular expression `/(a*)•(b+)•\1/`. This instrumentation differs from the one given in Figure 4.2 in that: **(1)** the character matcher combinators are wrapped inside calls to `saveState` and **(2)** it makes use of the combinator `saveStateBP` to save the matching states associated with the two break points.

Saved states can be of two types, *matching states* and *administrative states*. Matching states are those generated by the application of the regular expression combinators, while administrative states signal the points of the matching process where branching occurs and where the matching process is forced to backtrack due to a matching failure. We distinguish four types of matching states: *forward states*, *failure states*, *break point states*, and *epsilon states*. *Forward states* result from successfully matching either a character of the input string or a capturing group expression, respectively causing the matching index to advance and the captures array to be updated. *Failure states* signal a matching failure. *Break point states* signal a break point instruction. And, *epsilon states* signal the points where a quantified expression is matched against the empty string. Epsilon states are only used to facilitate the construction of the matching tree. Administrative states are used to bookkeep the information regarding branching and backtracking. Every time the matching process branches, it adds a *branch* administrative state to the matching trace. Analogously, every time the matching process backtracks, it adds a *back-track* administrative state to the matching trace. In the following, we depict forward states in white, failing states in red, branch states in dark blue, backtrack states in light blue, epsilon states in grey, and break point states in purple.

Both matching states and administrative states are internally represented as JavaScript objects with a property `id` storing the state's identifier and a property `type` storing the corresponding state type represented as a string (e.g. `"forward"`, `"breakpoint"`, `"failure"`, `"branch"`, `"backtrack"`, and `"epsilon"`). The property `id` is interpreted differently depending on the type of state. The `id` of a matching state corresponds to its unique identifier, while the `id` of an administrative state identifies the matching state with which it is associated, that being the state where the branching occurred. In the case of a branch state, the associated matching state is always the previous state, while in the case of a backtrack state, the associated matching state is the state to which the matching is supposed to jump.

While administrative states only have the properties `id` and `type`, matching states additionally define the following properties:

- **Initial** - [optional] Boolean - identifies the starting state;
- **Index** - Integer - tells us where we are in the input string, i.e., what index is currently being matched;
- **Loc** - Object - tells us what part of the regular expression is currently being matched, and its start and end indexes;
- **Inner Loc** - [optional] Object - identifies what part of a character range is being matched, for instance, in the expression `[a-zA-Z]`, it tells us if `a-z` or `A-Z` is being matched;
- **Captures** - Array - represents all captured strings up to the current state of the matching process;
- **Caps** - Integer - tied to the previous attribute, tells us how many captures exist in the entire matching process;
- **Str** - String - represents the initial input string.



```
▸0: {str: 'b', caps: 1, index: 0, captures: Array(1), id: 0, …}
▸1: {id: 1, type: 'branch'}
▸2: {str: 'b', caps: 1, index: 0, captures: Array(1), id: 1, …}
▸3: {id: 1, type: 'backtrack'}
▸4: {str: 'b', caps: 1, index: 0, captures: Array(1), id: 2, …}
```

```
▾2:
    caps: 1
  ▸captures: [undefined]
    id: 1
    index: 0
  ▸loc: {source: 'a', start: '1', end: '2'}
    msg: "characterSetMatcher"
    str: "b"
    type: "failure"
```

Figure 5.2: Trace generated when matching `a|b` against string `b`

In order to better understand how matching traces are constructed and represented inside REXSTEPPER, we will now consider three examples.

Example 1: Consider the matching of regular expression `a|b` against the string `b`. The generated matching trace is depicted in Figure 5.2 (a) as a JS object collection and in Figure 5.2 (c) as a diagram. In this example, we start by matching regular expression `a` against string `b` in state 1, which corresponds to a failure. Afterwards, a backtrack state with `id = 0` indicates that the matching process needs to backtrack to the matching state with the corresponding `id`. Finally, the right side of the disjunction, regular expression `b`, successfully matches string `b`, in state 2. In this example we can observe the behaviour of four types of states: forward; branch; backtrack; and failure.

Let us further detail one state of this trace, for instance state with `id` 1, depicted in Figure 5.2 (b):

- *Captures:* This state has no captures;
- *Index:* Has the value of 0, meaning it tried to match the first character of the input string;
- *Id:* Uniquely identifies this state with the number 1;
- *Loc:* Tells us which part of the regular expression is being matched, in this case it tells us we are matching the expression `a` which starts at index 1 and ends at index 2 of the regular expression;
- *Type:* Has the value `failure`, meaning it corresponds to a matching failure;

- *Caps:* Represents the global number of captures for the current match, which is 2;
- *Str:* Represents the initial input string being matched, b.

Example 2: Consider the matching of regular expression a+b against the string aab. The generated matching trace is depicted in Figure 5.3. In this example we introduce another type of matching state: the epsilon state. The matching process starts by successfully matching two as. Then, we can observe the typical sequence of states generated by greedily quantified expressions, i.e. a failure followed by a backtrack: it fails when trying to match a third a against the input string b, in state 3; and backtracks to a copy of state 2 by an epsilon transition, in state 4. Finally, in state 5, we end the matching process by successfully matching string b.

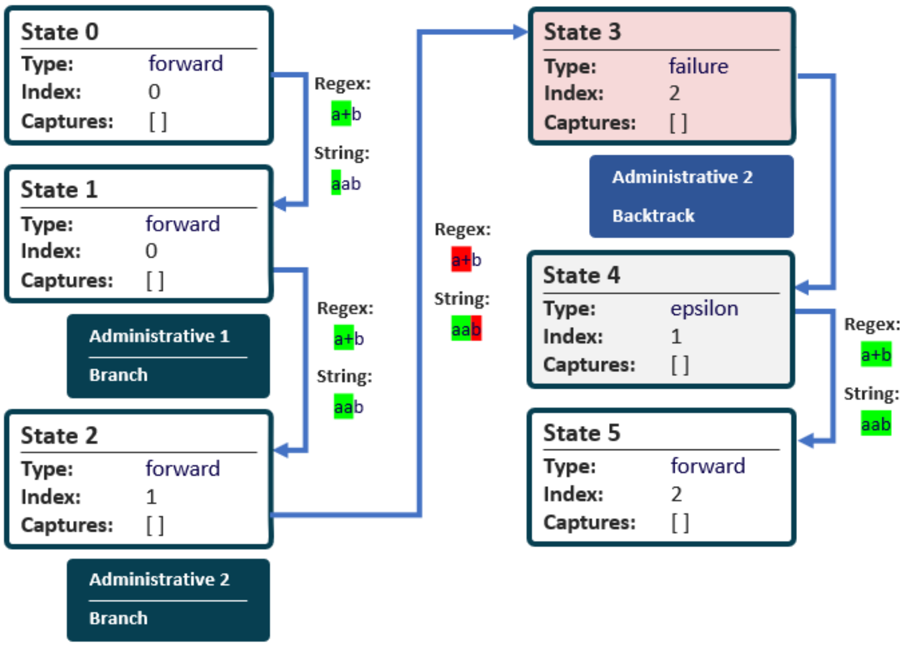


Figure 5.3: Trace generated when matching a+b against string aab

Example 3: Finally, consider the matching of regular expression (a*)\1b against the string aaab. The generated matching trace is depicted in Figure 5.4.

In this example, we can observe the second iteration of this particular matching process. Notice that the first Index to be matched is 1, and not 0, which would be the first index to consider. This is because when a given matching process does not find a match, it restarts that process from the next index available, in this case, index 1. This example starts by matching two as; failing to match a third one in state 22; and then backtracking to a copy of state 21 by an epsilon transition. It matches a capture group in state 24, updating the captures array to [aa] and proceeds to fail when trying to consume the backreference, which was expecting string aa but got character b. This causes the matching process to backtrack to a copy of state 20, which will then match another capture group, this time updating the

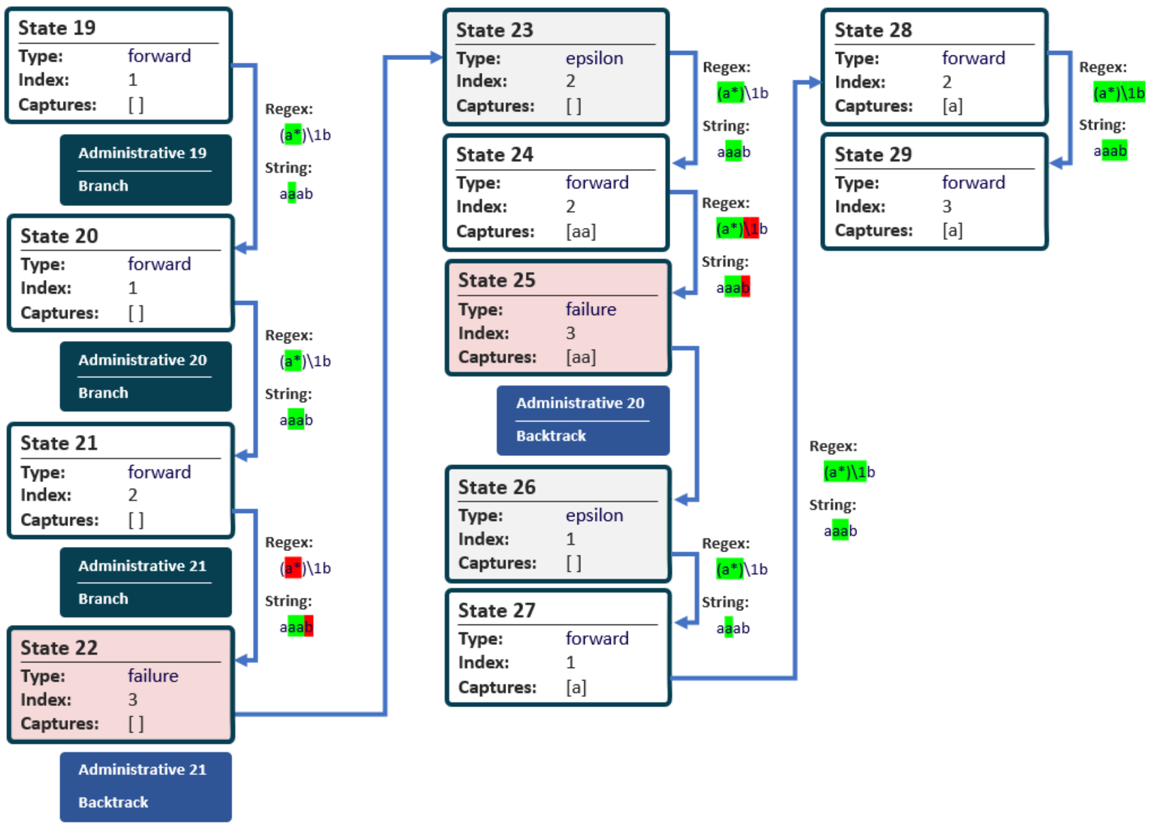


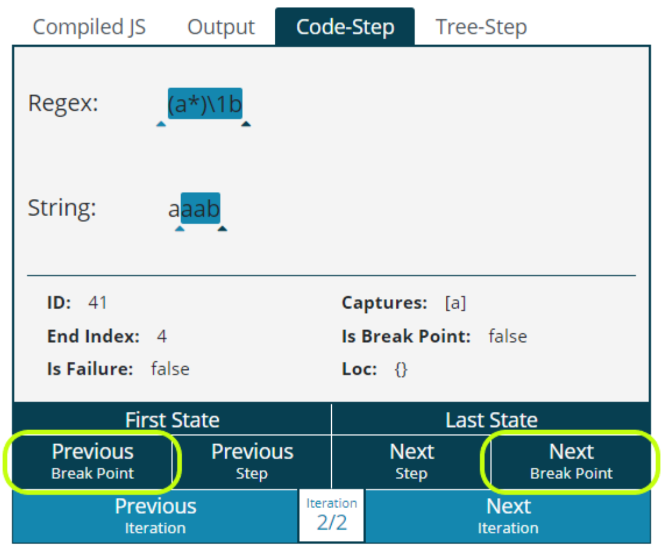
Figure 5.4: Trace generated when matching `(a*)\1b` against string `aaab`

captures array to `[a]` in state 27. Finally, it matches the backreference, which in this case is `a`, and the final character `b`.

5.3 Runtime Debugging

RexStepper receives as input the sequence of matching states produced by our instrumented version of RexRef and presents them visually to the developer. RexRef has two main trace visualisation modes: *code-stepping mode* and *tree visualisation mode*.

Code-stepping mode. In code-stepping mode the developer is shown a matching state at a time. Figure 5.5 depicts the RexStepper state inspection interface, which showcases: **(1)** the current matching index, **(2)** the captures array, **(3)** two boolean values respectively indicating if the current state corresponds to a break point or to a matching failure, and **(4)** the unique identifier of the depicted state. Furthermore, the global index, i.e. the index at which the current matching process started, is shown as the number of the current iteration. For instance, the matching state shown in Figure 5.5 corresponds to the second iteration of its respective regular expression match, for which the matching process started at index 1. Importantly, RexStepper highlights both the part of the input string that has been consumed so far as well as the part of the regular expression against which it was matched. Accordingly, the first `a` of the input string is not highlighted.



Regex: (a*)\1b

String: aaab

ID: 41
End Index: 4
Is Failure: false
Captures: [a]
Is Break Point: false
Loc: {}

Figure 5.5: REXSTEPPER Matching State Interface

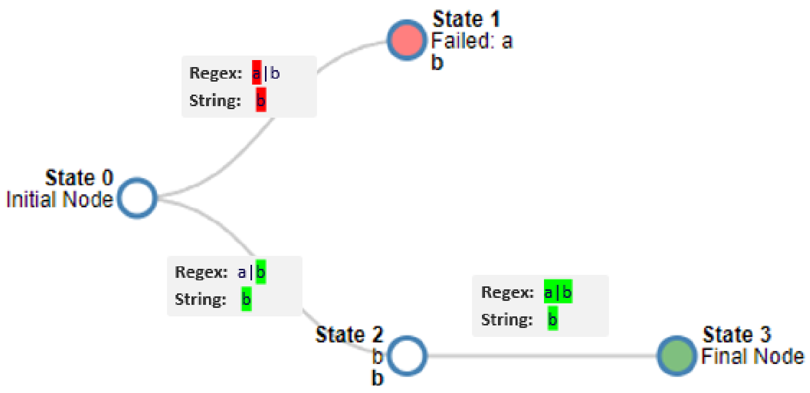


Figure 5.6: Tree generated for the match of `a|b` against the string `b`

Tree visualisation mode. It is often helpful to combine code-stepping with a global view of the entire the matching process. To achieve this, REXSTEPPER constructs a matching tree for each iteration of a regular expression match. For illustration purposes, we first constructed two trees from simple examples, namely for **(1)** matching the regular expression `a|b` against string `b`, in Figure 5.6; and for **(2)** matching the regular expression `a+b` against string `aab`, in Figure 5.7. These trees correspond to the traces presented in Figures 5.2 and 5.3, respectively. By inspecting the figures, we can clearly map states between both trace and tree representations. Some visual cues are shared between code-stepping mode and tree visualisation mode. For instance, white nodes continue to represent forward states and red nodes failure states. In contrast, epsilon states and break point states are only represented via text annotations. In order to improve the understandability of our tree examples, all transitions contain a grey box explaining what was matched in both the regular expression and the input string when transitioning from one state to another.

We depict a more complex example in Figure 5.8, which shows the matching tree generated for the second iteration of the match of `(a*)\1b` against the string `aaab`. This matching process was also rep-

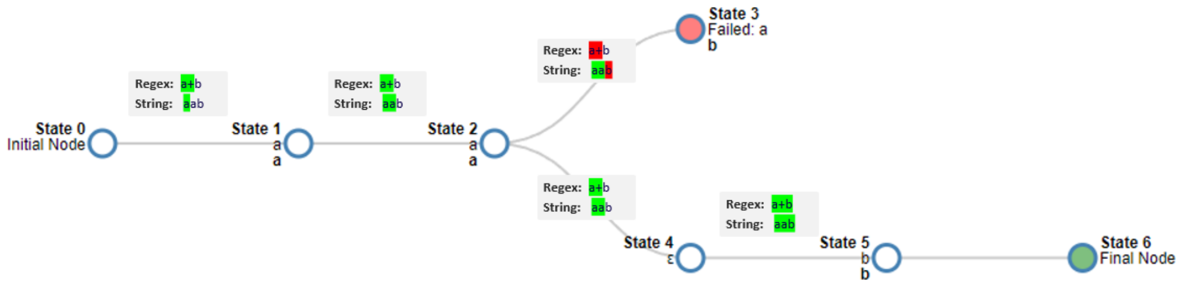


Figure 5.7: Tree generated when matching the regular expression `a+b` against the string `aab`

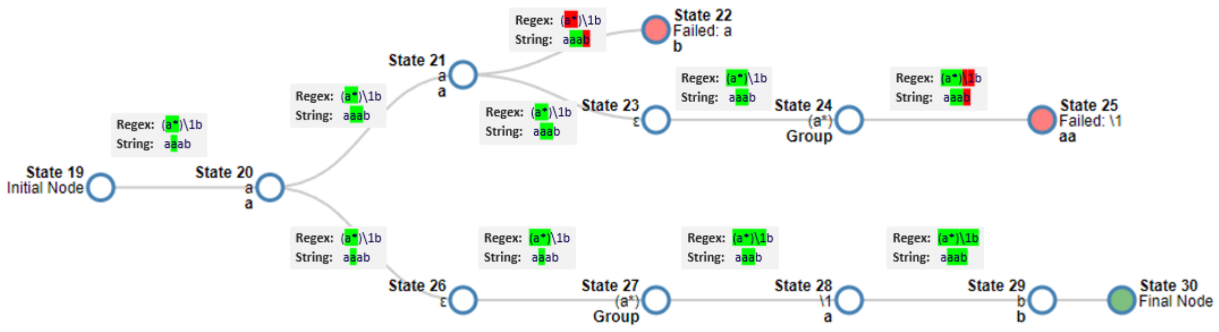


Figure 5.8: Tree generated for the second iteration of the match of `(a*)\1b` against the string `aaab`

resented as a trace before, in Figure 5.4. This tree clearly shows that the matching process backtracks two times before finding a successful match:

- *first backtrack (state 22):* the matcher tries to consume a third `a` and finds a `b`;
- *second backtrack (state 25):* the matcher consumes two `a`s, updates the first capture group to `aa`, and then tries to consume two `a`s again to match the backreference $\backslash 1$.

REXSTEPPER also generates the matching tree from the matching trace computed by REXREF. To this end, we include administrative states in the trace that signal the points of the matching process where branching occurs and where the matching process is forced to backtrack due to a failure.

We generate a matching tree using the auxiliary JavaScript library, D3 [53]. To this end, we first rearrange the matching trace for it to form a tree of state nodes, taking advantage of the administrative nodes created for this purpose. State nodes are simply wrappers around matching states, represented by the class `NodeD3` defined below.

```
class NodeD3 {
    constructor(name, node, arr) {
        this.name = name;
        this.node = node;
        this.children = arr;
    }
}
```

Now, we describe the algorithm that we use to convert a trace into a tree of state nodes, MAKETREE, which is depicted in the Algorithm 1. The algorithm is implemented recursively. The base case corresponds to the empty trace, which is mapped to the empty tree. In the recursive case, the algorithm proceeds as follows:

- It starts by separating the received trace in two elements: `head`, the first state of the given trace, and `tail` corresponding to the remaining states.
- Then, it searches for a backtrack state with the same `id` as the `head`. If true, this means that in the tree, this node will have two children, so we need to calculate the left and right children of this node as described below:
  - In order to do that, the algorithm gets the actual `id` number of that state and uses it to call the methods `left` and `right`, which will then divide the trace in two: the left side, `leftList`, will contain all states from the actual `head` to the backtrack with the same `id`; and the right side, `rightList`, will contain the nodes from that backtrack until the end of the trace.
  - Afterwards, the algorithm applies itself recursively on `leftList` and `rightList` in order to generate tree representations of those elements, obtaining `leftTree` and `rightTree`.
  - Finally, the algorithm returns a new node composed of the `head` node and its children, `leftTree` and `rightTree`.
- In case there is no backtrack state with the same `id` as the `head`, this means the current node will have only one child, so the algorithm just needs to recursively call itself on `tail`, obtaining `tree`; and returning a new node composed of the `head` and its only child `tree`.

Algorithm 1 Transform a trace into a tree

```
1: function MAKETREE($trace$)
2:     if $trace.length === 0$ then
3:         return $null$                                  ▷ If trace has no states, return empty array.
4:     end if
5:     $head, tail \leftarrow trace$
6:     if $existsBacktrackNodeWithId(head.id)$ then       ▷ If a Backtrack node with ID == head.ID exists
7:         $id \leftarrow getBacktrackNodeId(head.id)$                ▷ there is a bifurcation in this node.
8:         $leftList \leftarrow left(head, tail, id)$                      ▷ We calculate the left and
9:         $rightList \leftarrow right(head, tail, id)$                 ▷ right sides of the bifurcation;
10:        $leftTree \leftarrow makeTree(leftList)$        ▷ recursively call this function for each side;
11:        $rightTree \leftarrow makeTree(rightList)$
12:        $name \leftarrow$ ‘State$head.id‘
13:        return $node(name, head, [leftTree, rightTree])$        ▷ and return the ’head’ as a new Node.
14:    else              ▷ Otherwise, no bifurcation occurs, so this node contains exactly one child;
15:        $name \leftarrow$ ‘State $head.id‘
16:        $tree \leftarrow makeTree(tail)$      ▷ We calculate the child, recursively calling this function for the tail;
17:        return $node(name, head, [tree])$                ▷ and finally return the ’head’ as a new Node.
18:    end if
19: end function
```

Chapter 6

Evaluation

We evaluate RexRef and RexStepper separately. RexRef was tested against TEST262, the official JavaScript test suite, while RexStepper was used to debug 18 faulty regular expressions obtained from the Stack Overflow and Stack Exchange websites.

6.1 RexRef Evaluation

We show that RexRef is trustworthy by passing all applicable tests from Test262 [16], the official ECMAScript test suite. Test262 contains more than 35K tests, out of which 1893 target regular-expression-related functionality. From these tests, 729 tests target ES5; they are easily identifiable as they are labeled with the tag `es5id`. Out of these 729 tests, RexRef passes 718. The failing tests are not currently applicable to RexRef for reasons detailed below. The fact that we pass all the applicable tests, which constitute 98.49% of all ES5 regular-expression-related tests gives us a strong guarantee that our reference implementation is consistent with the behaviour described in the ECMAScript Standard. A breakdown of the testing results is presented in Table 6.1.

Category	Sub-category	Total	Compiled	Passed
RegExp	exec	61	61	60
RegExp	test	38	38	37
RegExp	others	28	28	28
String	match	37	37	35
String	replace	42	42	40
String	search	29	29	28
String	split	103	103	101
Matchers	_	391	389	389
Total		**729**	**727**	**718 (98,49%)**

Table 6.1: Test262 test suite results.

For clarity, we divide the tests into three main categories:

- `RegExp`: behaviour and internal representation of regular expressions, with emphasis on the built-in functions `exec` and `test`.
- `String`: behaviour of built-in String functions that interact with regular expressions: `match`, `replace`, `search`, and `split`.
- `Matchers`: other categories and sub-categories related to regular expression matchers.

We further subdivided these categories, providing for each sub-category the number of: **(1)** available tests in the test suite, **(2)** compiled tests, and **(3)** passing tests.

We classified 11 tests as not applicable: two fail to compile successfully, while others fail at execution time. We consider these tests not applicable (Table 6.2), as they make use of features currently unsupported by REXREF; in particular:

- REGEXPTREE [54], the regular expression parser used by REXREF, does not support *forward references*, which should match the empty string (onward referred to as **Parser Error**);
- REXREF only supports strict mode code as it uses ES6 modules, which automatically enforce strict mode even if the test is intended to be run in non-strict mode: for instance, strict mode causes the keyword `this` to be `undefined` in contexts in which it would otherwise be bound to the global object in non-strict mode (onward referred to as **Strict `this` keyword**) and does not allow for identifiers to be declared more than once in the same scope (onward referred to as **Strict Re-declaration**);
- REXREF does not throw an exception when built-in methods are invoked as constructors (e.g. `new String.split("abc", "b")`) (onward referred to as **Constructor Call Error**).

In order to extend REXREF to support the third category of non-applicable tests we would need to further instrument of the compilation, passing an extra argument to each method/constructor call to indicate whether or not the function is being called as a constructor. Given the limited scope and time frame of this thesis, we did not implement it.

Test file	Justification
RegExp/prototype/exec/S15.10.6.2_A1_T9 RegExp/prototype/test/S15.10.6.3_A1_T9	Strict Re-declaration
String/prototype/match/S15.5.4.10_A1_T3 String/prototype/replace/S15.5.4.11_A12 String/prototype/split/S15.5.4.14_A1_T3	Strict `this` keyword
String/prototype/match/S15.5.4.10_A7 String/prototype/replace/S15.5.4.11_A7 String/prototype/search/S15.5.4.12_A7 String/prototype/split/S15.5.4.14_A7	Constructor Call Error
RegExp/decimal-escape/S15.10.2.11_A1_T5 RegExp/decimal-escape/S15.10.2.11_A1_T7	Parser Error

Table 6.2: Not applicable tests

6.2 RexStepper Evaluation

To evaluate RexStepper, we composed a benchmark consisting of 18 faulty regular expressions obtained from the Stack Overflow and Stack Exchange forums. We divided the obtained regular expressions into five categories, each corresponding to a type of bug/misunderstanding. Table 6.3 summarizes our results, listing, for each category, the corresponding question ID. Stack Overflow IDs are prefixed with SO and Stack Exchange IDs with SE.

Category	Amount	Stack Overflow / Stack Exchange IDs
Assertion	5	SO_16472301, SO_18861, SO_49292762, SO_30441151, SO_32863970
Character Range	3	SO_47207164, SO_2211788, SO_53987537
Greediness	7	SE_54612, SO_40316670, SO_60142675, SO_37954914, SO_1413587, SO_37621986, SO_49671575
Groups	1	SO_12224711
Flags	2	SO_2851308, SO_1520800

Table 6.3: Categories of faulty regular expressions.

In the following, we explain each category together with its corresponding buggy regular expressions. For each analysed expression, we explain how to insert a break point that would lead directly to the error and how to identify the error from the inspection of the matching state at the break point.

6.2.1 Assertion bugs

The bugs in this category are mainly caused by developers misunderstanding the semantics of the `exec` method. Many developers ignore that `exec` tries to match the given regular expression starting at every index of the given input string. This leads them to omit the assertions ^ and $ when they want the supplied regular expression to match the entire input string instead of just a part of it. For instance, the regular expression `/\d+/` was used instead of `/^\d+$/` to match integer numbers. With RexStepper, developers can place a break point at the beginning of the regular expression, immediately understanding that the matching process would restart for each index of the given string. All faulty regular expression in this category are succinctly described in the tables below.

A1	
Regular expression used	`^[0-9]+$`
Correct regular expression	`^[0-9].*[0-9]$`
Expected match	Match any text that starts with at least one digit and ends with at least one digit.
Ideal break point placement	**`[!]`**`^[0-9]+$`
How can RexStepper help?	It clearly demonstrates that the expression `[0-9]+` is only matched once and the regular expression fails as soon as it tries to match characters other than numbers, because it is not taking them into account.

Table 6.4: Assertion Bug A1

A2	
Regular expression used	`^([a-zA-Z0-9_])+`
Correct regular expression	`^([a-zA-Z0-9_])+$`
Expected match	Match only numbers, letters and underscores.
Ideal break point placement	`^([a-zA-Z0-9_])+`**`[!]`**
How can RexStepper help?	It makes it clear that this regular expression is not matching the whole input string but just a part of it.

Table 6.5: Assertion Bug A2

A3	
Regular expression used	`\d+`
Correct regular expression	`^\d+$`
Expected match	Allow digits only.
Ideal break point placement	`\d+`**`[!]`**
How can RexStepper help?	It makes it clear that this regular expression is not matching the whole input string but just a part of it.

Table 6.6: Assertion Bug A3

A4
Regular expression used
`(https|http)?:\/\/(?:\w[\-\w.]+)(?:\/[\-\w+&@#\/%=~_|!:,.;]*)?(?:\?[\-A-Z0-9+&@#\/%=~_|!:,.;]*)?/i`
Correct regular expression
`^(https|http)?:\/\/(?:\w[\-\w.]+)(?:\/[\-\w+&@#\/%=~_|!:,.;]*)?(?:\?[\-A-Z0-9+&@#\/%=~_|!:,.;]*)?$/i`
Expected match
Validate whole URL’s.
Ideal break point placement
`(https|http)?:\/\/`**`[!]`**`(?:\w[\-\w.]+)(?:\/[\-\w+&@#\/%=~_|!:,.;]*)?(?:\?[\-A-Z0-9+&@#\/%=~_|!:,.;]*)?/i`
How can RexStepper help?
It makes it clear that just a part of the URL is being matched, it needs assertions to match entire URL’s.

Table 6.7: Assertion Bug A4

A5	
Regular expression used	`^[a-zA-Z]`
Correct regular expression	`^[a-zA-Z]+$`
Expected match	Alphabet characters only.
Ideal break point placement	`^`**`[!]`**`[a-zA-Z]`
How can RexStepper help?	It shows that an assertion is missing in order to match the whole string. As a secondary mistake, it would also show that there is a quantifier missing in order to match more than one character.

Table 6.8: Assertion Bug A5

6.2.2 Character Range bugs

The bugs in this category are mainly caused by developers ignoring the precise denotation of character ranges. For instance, the expression `/^[A-z0-9]+$/` was wrongly used instead of `/^[A-Za-z0-9]+$/` to match strings composed exclusively of alphanumeric characters. The problem is that the range `[A-z]` does not coincide with `[A-Za-z]`, additionally including various special characters whose ASCII codes lie between the code of `Z` and that of `a`. REXSTEPPER visually highlights the sub-range that is used to match each character of the sub-string, allowing the developer to easily identify this type of bug. All faulty regular expressions in this category are succinctly described in the tables below.

CR1	
Regular expression used	`^[A-z0-9]+$`
Correct regular expression	`^[A-Za-z0-9]+$`
Expected match	Alphanumeric characters only.
Ideal break point placement	`^`**`[!]`**`[A-z0-9]+$`
How can RexStepper help?	It shows that character `[` is being matched by range `A-z`, which does not coincide with `A-Za-z`, additionally including various special characters whose ASCII codes lie between the code of `Z` and that of `a`.

Table 6.9: Character Range Bug CR1

CR2	
Regular expression used	`[0-9]`
Correct regular expression	`.*[0-9].*`
Expected match	Strings with one digit.
Ideal break point placement	**`[!]`**`[0-9]`
How can RexStepper help?	It helps understanding that a character range only matches 1 character and not multiple ones.

Table 6.10: Character Range Bug CR2

CR3	
Regular expression used	`^[gG][o0O()\[\]{}][o0O()\[\]{}][gG][lLI][eE3]`
Correct regular expression	`[gG](?:[o0O]|\(\)|\[\]|<>){2}[gG][lLI][eE3]`
Expected match	The word Google, with characters that may resemble the shape of O's, L's and E's as their substitutes. O can be (), <>, [], , 0 and o; E can be e and 3; and L can be l and I. For instance, the string 'G()()gI3' would match.
Ideal break point placement	`^[gG]`**`[!]`**`[o0O()\[\]{}][o0O()\[\]{}][gG][lLI][eE3]`
How can RexStepper help?	It shows that when attempting to match the two `O`s, with special characters such as `(` and `)`, only one of them is being taken into account in the character range. For instance, the expression would only match `(` and not the corresponding `)`.

Table 6.11: Character Range Bug CR3

6.2.3 Greediness bugs

The bugs in this category are mainly caused by developers misunderstanding the semantics of greedy quantifiers when combined with the meta character $\cdot$. For instance, the expression `/Good .+\./` was wrongly used to match sentences starting with the word "`Good`" and ending with a period. This expression does not have the desired effect, since it matches any sequence of sentences such that the first word of the first sentence is "`Good`". For instance, instead of producing two separate matches for the string "`Good morning. Good afternoon.`", it produces a single match including both sentences. The fix is to forbid the matching of the period character before the end of the sentence: `/Good [^\.]+\./`. With RexStepper, the developer can place a break point before the $+$-quantifier to understand which characters are being matched before the final period. All faulty regular expressions in this category are succinctly described in the tables below.

GR1	
Regular expression used	`≪.+?IJ≫`
Correct regular expression	`≪[^≪]+?IJ≫`
Expected match	User wants to match strings enclosed by ≪ ≫ containing the characters `IJ` in the end, for instance `≪ABIJ≫`
Ideal break point placement	`≪`**`[!]`**`.+?IJ≫`
How can RexStepper help?	Our tool would show that expression `.+` is consuming ≪ and ≫ after the first ≪ match and the user should use `[^≪]+` instead

Table 6.12: Greediness Bug GR1

GR2	
Regular expression used	:.+\(.[^)]*$
Correct regular expression	:.+\([^)]+$
Expected match	Strings starting with : then any character, then (, then any character except).
Ideal break point placement	:.+\(**[!]**.[^)]*$
How can RexStepper help?	It would show that expression . is matching the character) which the user does not want to match.

Table 6.13: Greediness Bug GR2

GR3	
Regular expression used	Good .+\.
Correct regular expression	Good [^\.]+\.
Expected match	Sentences starting with "Good", followed by a space, other work and ending in a dot. For instance "Good Morning." or "Good afternoon.". The problem is this expression consumes strings like "Good morning. Good afternoon." as a whole match.
Ideal break point placement	Good **[!]**.+\.
How can RexStepper help?	It shows that greedy expression .+ consumes the actual dot that is supposed to be matched and all words that follow, up to the last dot of the input.

Table 6.14: Greediness Bug GR3

GR4
Regular expression used
^(?=.*?[A-Z])(?=.*?[a-z])(?=(.*?[0-9])|(.*?[@#&~])).{8,20}$
Correct regular expression
^(?=[^A-Z]*[A-Z])(?=[^a-z]*[a-z])(?=[^0-9@#&~]*[0-9@#&~])[A-Za-z0-9@#&~]{8,20}$
Expected match
String between 8 and 20 of length. Must contain 1 uppercase character, 1 lower case character and either at least 1 digit or at least 1 of these special characters @, #, &, ~.
Ideal break point placement
^**[!]**(?=.*?[A-Z])(?=.*?[a-z])(?=(.*?[0-9])|(.*?[@#&~])).{8,20}$
How can RexStepper help?
It shows that expression .* consumes every character, including most special characters (except for the new line).

Table 6.15: Greediness Bug GR4

GR5	
Regular expression used	`^%?\S{3}`
Correct regular expression	`^(%\S{3,})|((?!%)\S{3,})`
Expected match	String with a minimum of 3 characters. If `%` is the first character, that minimum changes to 4 characters.
Ideal break point placement	`^[!]%?\S{3}`
How can RexStepper help?	With an example string such as `%AB` users can see that the regular expression eventually backtracks when it does not find any match and consumes character `%` as part of expression `\S`, i.e., because `%?` means to "match character %, 0 or 1 times", it matches it 0 times in order to find a successful match.

Table 6.16: Greediness Bug GR5

GR6	
Regular expression used	`^[^;]*?variable2`
Correct regular expression	`^[^;\n]*?variable2`
Expected match	A variable name with the shortest match possible from the beginning of the given line. The problem is the resulting match also consumed the 2 lines before the one that contains the variable name (it undesirably consumes new lines).
Ideal break point placement	`^([^;][!])*?variable2`
How can RexStepper help?	Users can see that expression `[^;]*` is consuming new lines, resulting in a match that contains two extra lines, up until it reaches the desired variable name.

Table 6.17: Greediness Bug GR6

GR7	
Regular expression used	`\b(.*)\n*\s*\((\n*\s*.*\n*\s*)\)\n*\s*;`
Correct regular expression	`\b(.*?)\(([^)]*)\)\s*;\s*\n?`
Expected match	Match and capture function names and arguments. User had problems with multiple functions in the same line, which are matched as a whole.
Ideal break point placement	`\b(.*)\n*\s*\([!](\n*\s*.*\n*\s*)\)[!]\n*\s*;`
How can RexStepper help?	Users can understand that expression `.*` consumes the supposing end of the first match, which is character `;`.

Table 6.18: Greediness Bug GR7

6.2.4 Group bugs

This category deals with faulty expressions in which the developer does not make a consistent use of capture groups. Capture groups are used not only to further constrain the matching process via backreferences, but also in the context of the `replace` method. If a regular expression contains multiple capture groups at different nesting levels, it may be difficult to understand which ones correspond to the desired captures. In REXSTEPPER, developers can see all captures of the match as part of the current matching state. The faulty regular expression included in this category is described in the table below.

G1	
Regular expression used	`((\d+\.?\d+?)|(\d{1,3}(\,\d{3})+))*([a-zA-Z]+)`
Correct regular expression	`((?:\d+\.?\d+?)|(?:\d{1,3}(?:\,\d{3})+))*([a-zA-Z]+)`
Expected match	User is trying to match the specific input string 'Price: 123 dollar.' and he wants to capture the strings after 'Price:' but ends up capturing '123' in the two capture groups, instead of '123' and 'dollar'. The user is capturing unnecessary groups, ending up having too many captures and not being able to access the right ones via JavaScript.
Ideal break point placement	`((\d+\.?\d+?)|(\d{1,3}(\,\d{3})+))*([a-zA-Z]+)`**`[!]`**
How can RexStepper help?	Users can see which strings have been captured in each step of the match, effectively observing that too many groups were being captured at the end of the match.

Table 6.19: Group Bug G1

6.2.5 Flag Bugs

Faulty expressions in this category are related to the incorrect usage of flags. Most frequent mistakes have to do with the global flag, which developers misunderstand, thinking that it causes the expression to be matched multiple times within the same call to the `exec` method. Instead, it simply instructs the `exec` method to save the index corresponding to the end of the last computed match and to use that index as the starting index in the following call to `exec`. All faulty regular expressions in this category are succinctly described in the tables below.

F1	
Regular expression used	`/\d/g`
Correct regular expression	`/\d/`
Expected match	User is testing this expression against the same string two times with JavaScript's `test` method. His results alternate between `true` and `false`.
Ideal break point placement	`/`**`[!]`**`\d/g`
How can RexStepper help?	The `test` method internally calls `exec`, which is instructed by flag `g` to save the index corresponding to the end of the last computed match and to use that index as the starting index in the following call to `exec`. Users can clearly observe that by inspecting the second execution.

Table 6.20: Flag Bug F1

F2	
Regular expression used	`/FooB/gi`
Correct regular expression	`/FooB/`
Expected match	Similarly to the previous buggy expression, the user is testing his regular expression against the same string two times, with JavaScript's `test` method. His results also alternate between `true` and `false`.
Ideal break point placement	`/`**`[!]`**`FooB/gi`
How can RexStepper help?	The `test` method internally calls `exec`, which is instructed by flag `g` to save the index corresponding to the end of the last computed match and to use that index as the starting index in the following call to `exec`. Users can clearly observe that by inspecting the second execution.

Table 6.21: Flag Bug F2

6.3 Comparison with RegViz

In this section we compare RexStepper with RegViz, the academic tool that most resembles our debugger. In the following, we explain how to debug regular expressions with RegViz, the features that it offers, and its advantages and disadvantages when compared to RexStepper.

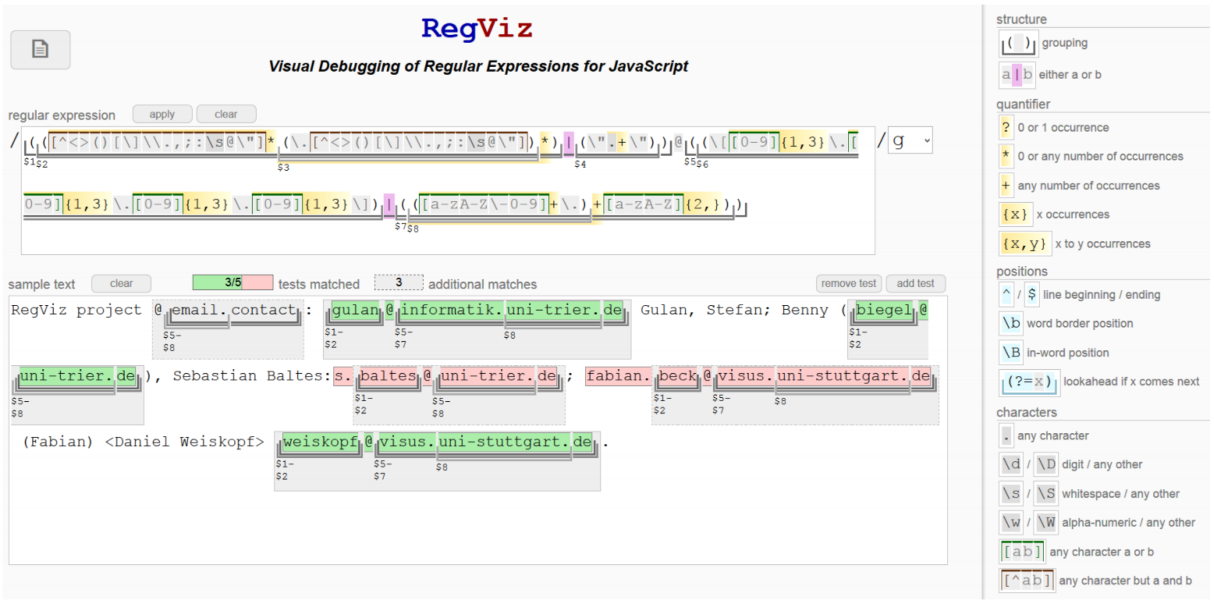
RegViz Overview RegViz is a JavaScript visual augmentation tool for regular expressions that aims at improving the readability of regular expressions by augmenting their syntax with extra visual cues instead of creating separate visual representations, such as finite state automata. RegViz offers visual augmentation of regular expressions by highlighting regular expression structural elements and colourizing special tokens. In particular:

- Capture groups are identified with horizontal lines below their corresponding expressions and are annotated with a unique identifier that maps directly to the respective captured string;
- Disjunctions are identified by placing their corresponding vertical slash character over a light purple background;

Other special tokens, such as quantifiers, anchors, and characters, are also highlighted with their own specific colours and/or lines.

RegViz supports user-defined tests, which are substrings of the sample text that should be matched by the given regular expression. Matched tests are marked with a green box, contrarily to the non-matched tests, which are marked with a red one.

The interface of RegViz, depicted in Figure 6.1, is divided into three sections: at the top, an editable text box shows the visually augmented expression; below, a text area where users insert the sample text they wish to match against the given regular expression; and lastly, at the right side, a cheat sheet explaining the colour codes used for each regular expression element.



```
/((([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))/g
```

```
RegViz project @email.contact: gulan@informatik.uni-trier.de Gulan, Stefan; Benny (biegel@
uni-trier.de), Sebastian Baltes:s.baltes@uni-trier.de; fabian.beck@visus.uni-stuttgart.de
 (Fabian) <Daniel Weiskopf> weiskopf@visus.uni-stuttgart.de.
```

Figure 6.1: REGVIZ example - regular expression that matches the generalization of e-mail addresses.

Comparison with REXSTEPPER In order to compare REXSTEPPER with REGVIZ, we analysed a real use case from REGVIZ's article [9], which is a regular expression that matches the generalization of e-mail addresses.

When debugging the expression, the visual cues implemented in REGVIZ, especially groupings (horizontal lines), are particularly helpful in order to visually understand such a complex expression as a whole, as it ends up slicing the "problem" into smaller pieces and making it easier to visually comprehend. Despite having visual augmentation and giving the user the ability to clearly identify that a given expression has a bug, it is still hard to pinpoint exactly where the actual bug is.

Regarding REXSTEPPER, its code-stepping capability and all the runtime information shown for each state offers a different kind of granularity when debugging complex expressions. Navigating between states gives the user the ability to identify the states with unexpected partial results.

One may think that REXSTEPPER is not practical when the regular expression is more complex or extensive, because users may have to step into hundreds of states before reaching a particular state they specifically want to debug. Breakpoints solve this problem, offering users the ability to jump directly to certain locations of the expression where they think a problem may arise and navigate between them with the help of the specific break point buttons, detailed in Section 5.1 and depicted in Figure 5.5.

Chapter 7

Conclusions

7.1 Conclusions

In this thesis, we studied a wide range of papers, covering four distinct topics: user studies; visualisation mechanisms; static analysers and regular expression synthesis. Summarizing these papers main findings, we conclude that: **(1)** there is a necessity to create better tools for writing and debugging regular expressions; **(2)** there is a lack of tools for detecting bugs in regular expressions; **(3)** regular expressions are not sufficiently tested in the wild; **(4)** graphical representations of regular expressions are more effective than their standard textual representations; and **(5)** synthesizing regular expressions can be an effective way of automatically fixing or generating regular expressions given a set of examples and/or descriptions.

We analysed 26 non-academic debugging tools, concluding that only three of them: REGEX 101, REGEX BUDDY and REGEX COACH offer the capacity of code-stepping regular expressions. These three tools have some limitations regarding that specific feature, mainly: some tools do not support the most used regular expression flavours; other tools do not represent which strings were captured by capture groups; and all of them are incapable of code-stepping regular expressions while maintaining their enclosing environment.

We have presented REXSTEPPER, the first regular expression code-stepper that allows for the debugging of ES5 regular expressions without taking them out of their enclosing JavaScript programs. We have built REXSTEPPER on top of REXREF, our novel reference implementation of ES5 regular expressions. REXREF was tested against TEST262, passing all the applicable tests, and REXSTEPPER was used to debug 18 real-world buggy regular expressions. REXREF and REXSTEPPER are Open Source code, available at the online repositories [28] and [27], respectively, for every developer to use.

7.2 Future Work

There are various action points we can cover in the future in order to improve REXSTEPPER. First of all, we plan to extend it with further debugging facilities, such as conditional break points, as well as syntactic visualisation mechanisms inspired by those introduced in REGVIZ [9]. Having REXREF support the next most recent version of the ECMAScript, ES12, would also be a considerable improvement. The new version of REXREF would have to be tested against the 1893 regular expression tests included in the most recent version of Test262. Finally, we would like to integrate REXSTEPPER with standard IDEs and browser engines. For instance:

- Deploying REXSTEPPER as a browser plug-in, where the plug-in itself would compile the JS code and the runtime environment would run on the console, with a simplified version of our debugging interface, offering the ability to visualize the match and navigate through the matching trace;
- Integrating REXSTEPPER with an existing IDE. This could be done by showcasing REXSTEPPER debugging facilities as a simplified interface on a side menu or as a split tab of the IDE. Then, every time the developer selected a regular expression, a right click option would trigger the debugger, copying the expression and string(s) to that interface and enabling the code-stepping and tree visualization modes.

Bibliography

[1] E. Gavrin, S. Lee, R. Ayrapetyan, and A. Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015*, pages 19–20. ACM, 2015.

[2] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. C. R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *16th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science. Springer, 2011.

[3] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS*.

[4] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*.

[5] M. Lee, S. So, and H. Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016*.

[6] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 487?502, New York, NY, USA, 2020. ACM.

[7] R. Pan, Q. Hu, G. Xu, and L. D'Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[8] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett. Sketch-driven regular expression generation from natural language and examples. *CoRR*, abs/1908.05848, 2019. URL `http://arxiv.org/abs/1908.05848`.

[9] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf. RegViz: visual debugging of regular expressions. In *36th International Conference on Software Engineering, ICSE '14,*.

[10] A. F. Blackwell. Swyn. In *Your Wish is My Command*, The Morgan Kaufmann series in interactive technologies.

[11] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2000*.

[12] regex101. Regular expressions 101. `https://regex101.com`.

[13] J. G. Software. Regexbuddy. `https://www.regexbuddy.com`.

[14] Weitz. The regex coach. `http://www.weitz.de/regex-coach/`.

[15] M. Sulzmann and K. Z. M. Lu. Fixing regular expression matching failure.

[16] ECMA TC39. Test262 Test Suite. https://github.com/tc39/test262, 2017.

[17] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A Trusted Mechanised JavaScript Specification.

[18] D. Park, A. Stefanescu, and G. Rosu. KJS: a complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[19] A. Charguéraud, A. Schmitt, and T. Wood. Jsexplain: A double debugger for javascript. In *Companion of the The Web Conference 2018 on The Web Conference 2018*. ACM, 2018.

[20] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In A. Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages*. ACM, 2012.

[21] J. F. Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner. Javert: Javascript verification toolchain. *Proc. ACM Program. Lang.*, (POPL), 2018.

[22] J. Park, J. Park, S. An, and S. Ryu. JISET: javascript ir-based semantics extraction toolchain. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 2020.

[23] J. F. Santos, P. Maksimovic, G. Sampaio, and P. Gardner. Javert 2.0: compositional symbolic execution for javascript. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010*.

[25] B. Loring, D. Mitchell, and J. Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*.

[26] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*.

[27] Anonymous. Rexstepper repository. `https://github.com/icst22sub36/icst22sub36.github.io`, .

[28] Anonymous. Rexref repository. `https://github.com/icst22sub36/RexRef`, .

[29] Anonymous. Rexstepper online tool. `https://icst22sub36.github.io/source_html/`, .

[30] C. Chapman, P. Wang, and K. T. Stolee. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 405–416. IEEE Press, 2017. ISBN 9781538626849.

[31] C. Chapman and K. T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 282–293, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931073. URL `https://doi.org/10.1145/2931037.2931073`.

[32] P. Wang and K. T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 668–678, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236072. URL `https://doi.org/10.1145/3236024.3236072`.

[33] J. C. Davis, D. Moyer, A. M. Kazerouni, and D. Lee. Testing regex generalizability and its implications: A large-scale many-language measurement study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 427–439, 2019. doi: 10.1109/ASE.2019.00048.

[34] N. Hollmann and S. Hanenberg. An empirical study on the readability of regular expressions: Textual versus graphical. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 74–84, 2017. doi: 10.1109/VISSOFT.2017.27.

[35] G. R. Bai, B. Clee, N. Shrestha, C. Chapman, C. Wright, and K. T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 197–208, 2019. doi: 10.1109/ICPC.2019.00039.

[36] J. C. Davis. On the impact and defeat of regular expression denial of service. 2020.

[37] S. C. Z.-X. Zheng, Li-Xiao; Ma and X.-Y. Luo. Ensuring the correctness of regular expressions: A review. 2021. doi: 10.1007/s11633-021-1301-4.

[38] . H. L. Andersson, A. Modernizing the syntax of regular expressions. 2020.

[39] C. A. Chapman. Usage and refactoring studies of python regular expressions. 2016.

[40] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 498–507. IEEE, April 2010. URL https://www.microsoft.com/en-us/research/publication/rex-symbolic-regular-expression-explorer-2/.

[41] S. H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA, 2006. ISBN 0763738344.

[42] J. R. S. T. S. H. Susan H. Rodger, Anna O. Bilska (Anya).

[43] E. Larson. Automatic checking of regular expressions. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 225–234, 2018. doi: 10.1109/SCAM.2018.00034.

[44] Regexlib. https://regexlib.com/.

[45] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(95)00182-4. URL https://www.sciencedirect.com/science/article/pii/0304397595001824.

[46] P. R. L. in Korea University. Alpharegex. https://github.com/kupl/AlphaRegexPublic.

[47] Microsoft. Z3 theorem prover. https://github.com/Z3Prover/z3.

[48] N. L. Nick Locascio. Deepregex. https://github.com/nicholaslocascio/deep-regex.

[49] P. P. Serge Toarca. Debuggex. https://www.debuggex.com/.

[50] Cyril. Cyrilex. https://extendsclass.com/regex-tester.html.

[51] J. C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6(3-4).

[52] D. P. Friedman and A. Sabry. CPS in little pieces: composing partial continuations. *J. Funct. Program.*, 12(6).

[53] M. Bostock. D3.js. https://github.com/d3/d3.

[54] D. Soshnikov. regexp-tree. https://www.npmjs.com/package/regexp-tree.