Code-Stepping Regular Expressions in the Browser

Luís Alberto Carvalho de Almeida

***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. 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.**

***Index Terms*—JavaScript, regular expressions, continuation-passing style, debuggers**

I. 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 [16]. 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 [31], [41].

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 [10], [17], [23], [25], 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 [4], [5], [19]. 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, 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 29 regular expression debugging tools[2] and discovered that only three come with code-stepping capabilities: REGEX 101 [29], REGEX BUDDY [36], and REGEX COACH [42]. When it comes to the debugging of JS regular expressions, 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 paper, 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 *breakpoints*, which allow the developer to navigate the control directly to the point (sic!) 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 facility.

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.

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

[2]A complete list of the screened tools can be found in [2].

At the core of RexStepper is RexRef, our new reference implementation of JS regular expressions. RexRef implements ECMAScript 5 [12] 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 [13], 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 [6], [9], [26]–[28], [32], 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 [24], [33]–[35] 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.

Contributions. In summary, our contributions are:

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

Replication package. The source code of both RexRef and RexStepper is available at [2]. The replication package additionally includes a Web interface [1] for interacting with RexStepper, which comes with the 18 analysed buggy regular expressions.

II. 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 [12] 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. Regular expressions $r \in \mathcal{RE}$, defined in the table below, include: character expressions, c; boundary assertion, ba; sequences of regular expressions, $r_1\, r_2$; disjunctions, $r_1 \,|\, r_2$; group expressions, $(r)_i$ and $(?\!:r)$; backreferences, $\backslash i$; positive and negative look-ahead expressions, $(?\!=r)$ and $(?!\,r)$; greedily and non-greedily quantified expressions, $r\,qt$ and $r\,qt\,?$; and positive and negative range expressions, $[\,rg\,]$ and $[\,\hat{}\,rg\,]$. We explain each of these classes of expressions in detail shortly, together with their corresponding matchers. The exposition assumes a basic understanding of regular expressions and their standard meaning; the unfamiliar reader is referred to [14] for a thorough introduction to the topic.

Syntax of JS Regular Expressions

$$\begin{aligned} r \in \mathcal{RE} \;&::= c \mid ba \mid r_1\, r_2 \mid r_1 \,|\, r_2 \mid (r)_i \mid (?:r) \mid \backslash i \mid (?=r) \\ &\quad\mid (?!\, r) \mid r\, qt \mid r\, qt\, ? \mid [\, rg\,] \mid [\, \hat{}\, rg\,] \\ qt \in \mathcal{QT} \;&::= \; ? \mid * \mid + \mid \{\, i\, \} \mid \{\, i,\, \} \mid \{\, i_1, i_2\, \} \\ rg \in \mathcal{CR} \;&::= c \mid c_1 - c_2 \mid rg_1 rg_2 \end{aligned}$$

A. Regular Expression Interpreter

Following the JS standard [12], RexRef evaluates regular expressions to regular expression *matchers*. Hence, a regular expression interpreter can be seen as a mathematical function `interp` $:: 2^{\mathcal{FL}} \rightarrow \mathcal{RE} \rightarrow \mathcal{M}$ 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 [15], [30] 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 a that state satisfies a given regular

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

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

Fig. 1. Matching states and capture groups

expression assertion; for instance, if the character at the current index in a non-word character. The table below summarises our regular expression interpretation domains.

Interpretation Domains

MATCHERS	$m \in \mathcal{M}$	::	$\Sigma \rightarrow \mathcal{K} \rightharpoonup \mathcal{O}$
TESTERS	$t \in \mathcal{T}$	::	$\Sigma \rightarrow \mathcal{B}ool$
CONTINUATIONS	$\kappa \in \mathcal{K}$	::	$\Sigma \rightarrow \mathcal{O}$
OUTCOMES	$o \in \mathcal{O}$	::=	$\mathbf{S}\langle\sigma\rangle \mid \mathbf{F}\langle\sigma\rangle$
STATES	$\sigma \in \Sigma$	$\triangleq$	$\mathcal{S}tr \times [\mathcal{S}tr_\bot] \times \mathcal{I}nt$
STRINGS	$s \in \mathcal{S}tr_\bot$	$\triangleq$	$\mathcal{S}tr \uplus \{\mathsf{undefined}\}$
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 input string on which the outermost matcher was called, s, **(2)** an internal array $\vec{s}$, called *captures array*, mapping capture group indexes to their corresponding captures, and **(3)** the index of the input string to be processed next, i. To better understand how the captures array works, let us consider the two examples given in Figure 1.

In the left-hand-side example, we match the regular expression `/(a+)(b*)(c*)/` against the string `aabbbbccc` (ignore underlined characters for now). This expression has three capture groups, each corresponding to a different parenthesised subexpression. During the matching, 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. The figure 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, our matching states expose the following methods for accessing/updating their components:

- σ.getS() obtains the input string on which the outermost matcher was called;
- σ.getIdx() and σ.setIdx(i) obtain and update the current matching index, respectively;
- σ.getCap(i) and σ.setCap(i, s) obtain and update the i-th capture of σ, respectively;
- σ.getCaps() and σ.setCaps($caps$) obtain a deep copy of the captures array and update its value; and
- σ.copy() returns 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 [15], [30], 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 $r_1 | r_2$ 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)) {
      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 ^ and $ respectively denote the beginning and the end of the input string. Boundary assertions are evaluated to testers functions, 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. 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 $(0, 1)$.

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.

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

Fig. 2. Transpiled JS Program

```
function gRepeat (m, min, max) {
  return m_new(st, cont) {
    if (max == 0) { return cont(st) }
    var cont_d = function (st_d) {
      var 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 for space reasons.

B. 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 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 above. 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.

C. 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

```
// 15.10.6.2 RegExp.prototype.exec(string)
function exec (string) {
  ...
  // 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)) {
       // Let i = i+1.
       i = i+1
     // d. else
     } else {
       // i. Let r be the State result of the call
       ↪  to [[Match]].
       var r = ret.__State__;
       // ii. Set matchSucceeded to true.
       matchSucceeded = true
     }
  }
...
}
```

Fig. 3. `RegExp.prototype.exec(string)`

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 [6].

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

```
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);
}
```

Fig. 4. `__match__` wrapper

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 given in Figure 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, is just a 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*.

III. 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 *breakpoints*, which allow the developer to quickly move the control to the point where they think a bug might originate. 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 breakpoint; and **(4)** *multi forward step*, to move the control to the matching state at the next breakpoint.

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 input. In the browser, however, user interaction happens mostly asynchronously, with the exception of only 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 breakpoints, runtime instrumentation, and debugging facilities.

A. Breakpoints

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

$$r^{\bullet} \in \mathcal{RE}^{\bullet} ::= r \in \mathcal{RE} \mid \bullet \qquad (1)$$

In the online tool, we use the sequence of characters `[!]` instead of the symbol • to denote a breakpoint. Breakpoints 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 breakpoint, they will be shown the matching state after the expansion of `(b+)` is recognised.

B. 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 σ.save() and σ.saveBP() to save the current state for later use at debugging time. The main difference between these two methods is that the σ.saveBP() is used specifically to save intermediate matching states associated with breakpoints. Extended states expose various other methods that will be introduced by need.

Bookkeeping matching combinators. We extend the matching combinators introduced in §II-A 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)
 } }
```

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

Fig. 5. Transpiled Regular Expression with Breakpoint

```
function saveStateBP() {
  return function(st, cont) {
     st.saveBP(cont); return cont(st)
} }
```

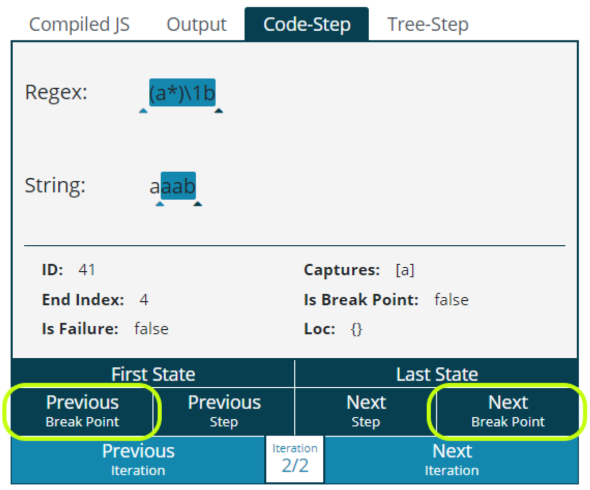
These combinators are simply wrappers around the corresponding extended state methods described above. The first one, `saveState`, returns a new matcher that starts by creating a wrapper continuation that saves the state produced by matcher `m` before calling the supplied continuation, and then calls 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`.

Instrumentation. The REXREF transpiler discussed in §II-B 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 changes. Hence, instead of wrapping all intermediate matchers inside a `saveState` combinator, we only wrap the matchers corresponding to: characters, character ranges, and backreferences. For instance, Figure 5 shows the stylised compilation of the regular expression `/(a*) • (b+) • \1/`. This instrumentation differs from the one given in Figure 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 breakpoints.

C. 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 6 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



Regex: (a*)\1b

String: aaab

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

Fig. 6. REXSTEPPER Matching State Interface

corresponds to a breakpoint 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 6 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.

Tree visualisation mode. It is often helpful to combine code stepping with a global view of entire the matching process. To achieve this, REXSTEPPER constructs a matching tree for each iteration of a regular expression match. Figure 7 shows the matching tree generated for the second iteration of the match of /(a∗)\1b/ against the string aaab. 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 \1.

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

IV. EVALUATION

We evaluate REXREF and REXSTEPPER separately. REXREF was tested against TEST262, while REXSTEPPER was used to debug 18 faulty regular expressions obtained from the Stack Overflow and Stack Exchange websites.

A. REXREF: *Correctness*

We show that REXREF is trustworthy by passing all applicable tests from Test262 [13], 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 I. 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.

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 I
TEST262 TEST SUITE RESULTS.

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

- REGEXPTREE [37], the regular expression parser used by REXREF, does not support *forward references*, which should match the empty string;
- 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;

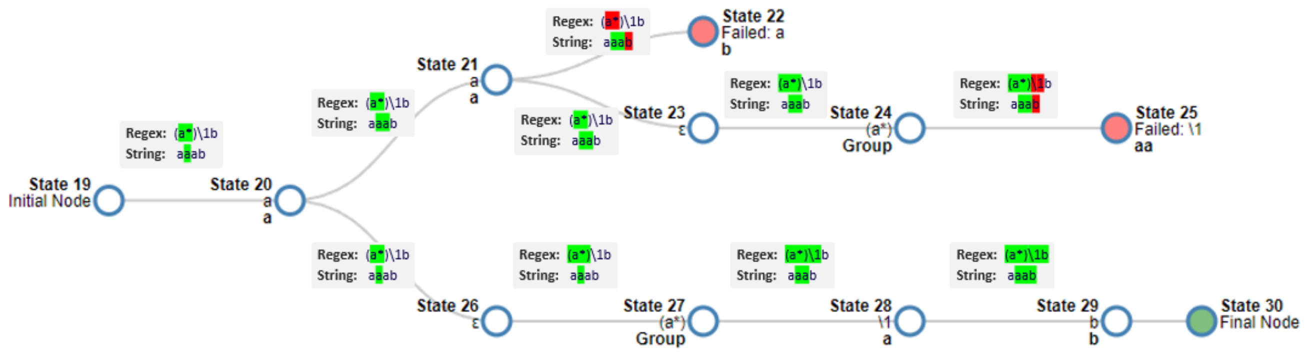


Fig. 7. Tree generated for the second iteration of the match of `/(a*)\1b/` against the string `aaab`

- REXREF does not throw an exception when built-in methods are invoked as constructors (e.g. `new String.split("abc", "b")`).

We could extend REXREF to support the third category of non-applicable tests. However, this would require further instrumentation 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. For the moment, we judge that the added complexity would outweigh the benefits.

B. REXSTEPPER: *Applicability*

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 II summarizes our results, listing, for each category, the corresponding Question IDs. 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 II
CATEGORIES OF FAULTY REGULAR EXPRESSIONS.

Below we give a high-level description of the proposed categories, illustrating the most relevant ones with a faulty expression from Table II. The IDs of the explained faulty expressions are underlined in the table, while a complete account of all faulty expressions can be found in the online repository [2]. 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.

Assertion. 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 breakpoint at the beginning of the regular expression, immediately understanding that the matching process would restart for each index of the given string.

Character Range. 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.

Greediness. The bugs in this category are mainly caused by developers misunderstanding the semantics of greedy quantifiers when combined with the meta character ·. 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 breakpoint before the +-quantifier to understand which characters are being matched before the final period.

Groups. 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 trivially see all captures of the match as part of the current matching state.

Flags. 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`.

C. Threats to Validity

A potential threat to internal validity is that, due to the complexity of the proposed tool, implementation bugs may remain somewhere in the codebase. We extensively tested the tool to mitigate this risk. Furthermore, the tool and the raw data are publicly available for other researchers and potential users to check the validity of the results.

A potential threat to external validity is the fact that the set of regular expressions collected from Stack Overflow and Stack Exchange websites may not be an accurate representation of those that occur during development. We try to reduce the selection bias by understanding the context in which the regular expression is being used. We aim at reducing threats to external validity and ensure the reproducibility of our evaluation by providing the source of the tool, the scripts used to run the evaluation, and all data gathered.

V. Related Work

There is vast body of research on regular expressions, covering topics as varied as: symbolic execution for regular expressions [24], [34], regular expression synthesis [10], [17], [23], [25], visualisation mechanisms [4], [5], [19], and user studies [3], [7], [8], [11], [18], [40]. In the following, we focus our analysis of the related work on: visualisation mechanisms, user studies, and parsing combinators. The reader is referred to [43] for a recent broad-spectrum survey on techniques for ensuring the correctness of regular expressions.

User Studies. There is a vast number of empirical studies aimed at characterising how regular expressions are (mis)used in practice [3], [7], [8], [11], [18], [40]. These studies tackle a wide variety of questions regarding the pragmatics of regular expressions, such as: **(1)** How often are regular expressions used by typical programmers? [7] **(2)** What regular expression patterns hinder understandability? [8] **(3)** What type of debugging infrastructure is more effective when it comes to finding errors/understanding regular expressions? [18] **(4)** What type of tools and techniques developers employ when having to write regular expressions? [3] and **(5)** How well are regular expressions tested in practice? [40]. Although none of these studies target the use of regular expressions in the context of JavaScript applications, we believe their findings to be indicative. For instance: the authors of [7] find that 50% of Python programmers make use of regular expressions at least once a week; the authors of [40] show that the vast majority of the regular expressions used in Java projects on GitHub is not properly tested; and the authors of [18] observe that developers tend to prefer visual debugging mechanisms rather than textual ones. These findings reinforce our view that developers need better debugging tools for writing their regular expressions.

Visualisation Mechanisms. Several research projects have tackled the problem of providing visual representations for regular expressions [4], [5], [19]. 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 [5], [19] and those that augment the syntax of the given regular expressions with extra visual annotations [4]. 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 mode*, REXSTEPPER provides an entirely new diagram (the matching tree) that explains the matching process.

Parsing Combinators. The use of combinators for parsing has a long tradition in the Functional Programming community [20], [21], [38]. The matching combinators presented in this paper can be seen as a refactoring of the regular expression specification that comes as part of the JavaScript standard [12]. However, we believe that they are easier to understand than the JS specification, and, therefore, accessible to a wider audience. Both the JS standard and our matching combinators have at their core the matcher type. This type can be seen as a combination of the traditional parser monad [22] and the continuation monad [39]. The establishment of a formal correspondence between these types is, however, left for future work.

VI. Conclusions and Further Work

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.

In future, we plan to extend REXSTEPPER with further debugging facilities, such as conditional breakpoints, as well as syntactic visualisation mechanisms inspired by those introduced in RegViz [4].

References

[1] Anonymous. Rexstepper online tool. https://icst22sub36.github.io/source_html/.

[2] Anonymous. Rexstepper repository. https://github.com/icst22sub36/icst22sub36.github.io.

[3] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019*. IEEE / ACM, 2019.

[4] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. RegViz: visual debugging of regular expressions. In *36th International Conference on Software Engineering, ICSE '14,*.

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

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification.

[7] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017. IEEE Press, 2017.

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

[10] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil 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.

[11] James C. Davis, Daniel Moyer, Ayaan M. Kazerouni, and Dongyoon Lee. Testing regex generalizability and its implications: A large-scale many-language measurement study. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[12] ECMA TC39. The 5th Edition of the ECMAScript Language Specification. Technical report, ECMA, 2011.

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

[14] Jeffrey E.F. Friedl. *Mastering Regular Exxpressions*. O'Reilly, 2006.

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

[16] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey 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.

[17] Sumit 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*.

[18] Niklas Hollmann and Stefan Hanenberg. An empirical study on the readability of regular expressions: Textual versus graphical. In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2017.

[19] Ted Hung and Susan 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*.

[20] Graham Hutton. Parsing using combinators. In Kei Davis and John Hughes, editors, *Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989, Fraserburgh, Scotland, UK*, Workshops in Computing, pages 353–370. Springer, 1989.

[21] Graham Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992.

[22] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4), 1998.

[23] Mina Lee, Sunbeom So, and Hakjoo 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*.

[24] Blake Loring, Duncan Mitchell, and Johannes 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*.

[25] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[26] Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: a complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

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

[28] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In Alessandro Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages*. ACM, 2012.

[29] regex101. Regular expressions 101. https://regex101.com.

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

[31] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS*.

[32] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. Javert: Javascript verification toolchain. *Proc. ACM Program. Lang.*, (POPL), 2018.

[33] José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. Javert 2.0: compositional symbolic execution for javascript. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[34] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010*.

[35] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon 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*.

[36] Just Great Software. Regexbuddy. https://www.regexbuddy.com.

[37] Dimitry Soshnikov. regexp-tree. https://www.npmjs.com/package/regexp-tree.

[38] S. Doaitse Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008*, volume 5520 of *Lecture Notes in Computer Science*, pages 252–300. Springer, 2008.

[39] Philip Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992.

[40] Peipei Wang and Kathryn 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, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn 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.

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

[43] Lixiao Zheng, Shuai Ma, Zuxi Chen, and Xiangyu Luo. Ensuring the correctness of regular expressions: A review. *Int. J. Autom. Comput.*, 18(4), 2021.