TreeRegex: An Extension to Regular Expressions for Matching and Manipulating Tree-Structured Text (Technical Report)

Benjamin Mehne

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-202
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-202.html

December 12, 2017

TreeRegex: An Extension to Regular Expressions for Matching and Manipulating Tree-Structured Text (Technical Report)

Benjamin Mehne
University of California, Berkeley
bmehne@cs.berkeley.edu

ABSTRACT

Tree-structured text is ubiquitous in software engineering and programming tasks. However, despite its prevalence, users frequently write custom, specialized routines to query and update such text. For example, a user might wish to rapidly prototype a compiler for a domain-specific language by issuing successive transformations, or they might wish to identify all the call sites of a particular function in a project (e.g. `eval` in JavaScript). We propose a natural and intuitive extension to regular expressions, called `TreeRegex`, which can specify patterns over tree-structured text. A key insight behind the design of `TreeRegex` is that if we annotate a string with special markers to expose information about the string's tree structure, then a simple extension to regular expressions can be used to describe patterns over the annotated string. We develop an algorithm for matching `TreeRegex` expressions against annotated texts and report on five case studies where we find that using `TreeRegex` simplifies various tasks related to searching and modifying tree-structured texts.

ACM Reference format:
Benjamin Mehne. 2017. `TreeRegex`: An Extension to Regular Expressions for Matching and Manipulating Tree-Structured Text (Technical Report). In *Proceedings of , , ,* 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Tree-structured text is widely used in many different software engineering and programming tasks. Examples of tree-structured text include various programming languages, domain-specific languages, and data formats such as XML and JSON. Users of tree-structured text often need to query patterns over such text and modify text. For example, a user may want to query if the `eval` function has been called in the body of any function in a JavaScript program or rapidly prototype a compiler for a domain-specific language by modifying the abstract-syntax tree of a program.

There are several languages and associated tools that users often use to search patterns and to modify tree-structured text. Regular expressions [34, 35, 38, 52, 56] are one such special formalism that is used to describe text patterns. They are widely used by programmers and computer scientists [28, 36, 66] to concisely and elegantly describe search patterns over text. Most popular programming languages support regular expressions; some popular text-processing languages, such as `awk`, `sed`, and `perl`, were designed around regular expressions. A key reason behind the popularity of regular expressions is that they are compact and concise. Moreover, regular expressions can extract substrings from texts. This is particularly useful in extracting information and in modifying texts.

However, formal regular expressions are not expressive enough to describe patterns over text having a tree-like structure. For example, it is impossible to write a regular expression that matches a block of statements in a C-like language because a statement block can have nested blocks.

Context-free grammars (CFGs) overcome the limitations of regular expressions by providing a more expressive formalism for describing patterns over tree-structured text. Although CFGs are strictly more powerful than regular expressions, they are not as compact and concise as regular expressions—pattern matching and replacing with CFGs, naïvely, requires the user to write an explicit program.

Term-rewriting systems [15, 18, 20, 22, 24, 51, 58, 61] simplify the use of CFGs for text search and modification. These systems allow programmers to describe tree rewriting declaratively as a set of rules. Though term-rewriting systems have been found to be more convenient to use compared to traditional parsers and abstract-syntax tree (AST) visitors, they require a complete porting of the CFG of the given text to the term-rewriting system, which could be non-trivial for complex languages.

We propose a natural and intuitive extension to regular expressions, called `TreeRegex`, which can specify patterns over tree-structured text. A key insight behind the design of `TreeRegex` is that *if we annotate a string with special markers to expose information about the string's tree structure, then a simple extension to regular expressions can be used to describe patterns over the annotated string.* Based on this insight, we propose a two-step process for specifying and matching `TreeRegex` expressions against a tree-structured string. In the first step, we annotate the string by inserting parenthesis meta-characters ($(^{\%}$ and $_{\%})$) in the string[1]. This annotated string, which resembles an S-expression in LISP [53], has *balanced* occurrences of ($(^{\%}$ and $_{\%})$) and is called a *serialized tree*. For example, $(^{\%}2+(^{\%}3*4_{\%})_{\%})$ is the annotated string for $2+3*4$. The parenthesis meta-characters make the tree-structure of the string explicit. An

[1] In the implementation, the plain ASCII parentheses and percent characters are used. We use a sub- and super-script here to improve legibility in these and related characters.

existing parser and source-code generator (i.e., a tool that serializes an abstract-syntax tree to the original string) could easily be modified to generate an annotated string.

In the second step, we write patterns over the serialized tree as `TreeRegex` expressions. A `TreeRegex` expression is a regular expression extended with balanced ($(^{\%}$, $_{\%})$), balanced ($(^{*}$, $_{*})$), and the wildcard meta-character **@**. The exact motivation and semantics of these extra meta-characters are described in the following two sections. After describing a pattern in `TreeRegex`, we match the pattern against the serialized tree using an efficient implementation of our algorithms, called `TreeRegexLib`.

`TreeRegex` has several key advantages. 1) It nicely decouples the CFG and parsing aspect of a string from the pattern expression and matching aspect. One can easily modify an existing parser in the first step to create a serialized tree—there is no need to port the CFG to our system. 2) `TreeRegex` is a natural extension to regular expressions, which we believe would be easy to learn if one is familiar with regular expressions. 3) Since `TreeRegex` is independent of the underlying CFG used to generate the serialized trees, it can be used to replace a sub-tree in a serialized tree with a sub-tree or string that does not conform to the original CFG. We take advantage of this flexibility in a case study that generates MIPS [3] assembly code from a simple language, based on the BC [2] calculator language. For this compilation, our approach requires no information about the grammar of MIPS. 4) One can restore the original string from a serialized tree by dropping the parenthesis meta-characters ($(^{\%}$ and $_{\%})$). This becomes useful while debugging `TreeRegex`. 5) `TreeRegex` matching and replacement algorithms can be implemented easily by using the API of an existing regular expression library. Thus, `TreeRegex` can be easily ported to many languages and this will allow programmers to use `TreeRegex` with their favorite languages to deal with tree-structured texts. So far, we have implemented `TreeRegex` for C++, Java, and JavaScript programs as `TreeRegexLib`, we have released the C++, the most mature version here: https://treeregexlib.github.io.

We apply `TreeRegexLib` to five case studies: a tool for instrumenting JavaScript programs for branch coverage, a tool to prevent SQL injection vulnerabilities, a linter for JavaScript, a tool for finding errors in C programs, and a compiler from a BC-like [2] language to MIPS assembly code. In our case studies, we found `TreeRegex` to be powerful enough for our tasks. We also found that we wrote significantly fewer lines of code while using `TreeRegexLib` compared to conventional AST-based techniques. Our experiments on the compiler for the BC-like language show that our `TreeRegexLib` implementation runs fast enough for practical usage—we can compile a 160kB file in less than 1 second.

2 OVERVIEW

We gently introduce `TreeRegex` through a series of motivating examples that we often encounter in program analysis and compiler construction.

Motivating Example. Suppose we want to check if the function `eval` has been called inside the body of any function in a JavaScript program[2]. We may try to find such a usage of `eval` using a regular expression. The following expression comes to mind[3]:

```
function .*(.*){.*eval(.*).*}
```

where `.` matches any character and `*` is the Kleene star operator (we do not treat parentheses as a meta-character here). Unfortunately, this regular expression does not work—it will match the following code, for instance:

```
function f1(x){bar()} eval(s); function f2() {}
```

This is because `.*` will match too much. Making the following slight modification to the regular expression prevents this problem:

```
function .*(.*){[^}]*eval(.*).*}
```

Here, we use `[^}]`, a character class excluding curly braces, instead of the `.` meta-character. Unfortunately, this regular expression does not match

```
function f1(v){{bar()}eval(s)}
```

which should be matched—it has a call to `eval` in its body, preceded by a block with a call to `bar`. Both of these example regular expressions fail because they ignore the structure of the target expressions. To find the pattern we are looking for, we must specify that `eval` can appear either at the top-level block or in some nested block of a function body.

Patterns over structured text could be expressed using context-free grammars (CFGs) [29]. A standard technique to search for such patterns is to write a full-fledged CFG of the JavaScript language and then use a parser to convert a JavaScript program into an abstract-syntax tree (AST). The code pattern can then be searched by performing a programmatic traversal of the AST. This is the de-facto technique that various linters use to discover problematic code snippets. Unfortunately, this technique has a few disadvantages. First, we need to understand the structure of the AST enough to know where to look for the definition of a function and for the invocation of the `eval` function. Second, searching for a particular code pattern requires us to write a program that visits over the AST and explicitly looks for the identifier `eval` in a function definition sub-AST. Such code will span several lines and will not be as compact as a simple, single-line regular expression.

2.1 TreeRegex and serialized trees

We propose a two-step technique to represent strings having tree-like structure and to express search patterns over them. In the first step, we convert a string into an annotated string which makes the tree structures of the string explicit. Specifically, we convert a string into an annotated string, similar to S-expressions in LISP [53], where the recursive structures are surrounded by the special parenthesis meta-characters ($(^{\%}$ and $_{\%})$). For example, we convert the string "$3 * 4 + 5 * 6$", denoting an arithmetic expression, into the annotated string "$(^{\%}(^{\%}3 * 4_{\%}) + (^{\%}5 * 6_{\%})_{\%})$".[4] Such an annotated string has two important properties:

[2]We are interested in this particular code pattern because calling the `eval` function inside a JavaScript function 1) prevents just-in-time (JIT) compilation of the function, and 2) can unexpectedly change the local variables of the enclosing function, which makes reasoning about the correctness of the function difficult.

[3]Whitespace handling is ignored in this section for simplicity of exposition.

[4]In this serialization we do not enclose the integer literals in ($(^{\%}$ and $_{\%})$) to simplify exposition and to reduce clutter. In our actual implementation, we surround integers with ($(^{\%}$ and $_{\%})$).

- An annotated string has balanced parentheses ($(^{\%}$ and $_{\%})$). That is, each opening parenthesis ($(^{\%}$ has a later corresponding closed parenthesis $_{\%})$), and the string between the pair of parentheses is again balanced.
- In an annotated string, if we remove the parenthesis meta-characters, we get back the original string.

The string between a pair of balanced parentheses denotes a structure that can have other nested structures. We call such annotated strings *serialized trees*. To convert a string into a serialized tree, one can use an existing parser and an AST-to-source code generator. Programming languages and various structured data formats, such as XML, usually come with an implementation of a parser and an AST-to-source code generator, and they could be easily modified to annotate a string with ($(^{\%}$ and $_{\%})$). For example, we modified 108 out of 2298 lines of code in `esotope` code generator [10] to generate serialized trees for JavaScript programs. We have also implemented a generic serialized tree generator for ANTLR [47] grammars using 128 lines of Java code. A key advantage of converting a string into a serialized tree is that a simple extension to regular expressions can now be used to describe patterns over serialized trees.

The second step of our technique will be to check whether an annotated input string matches a desired pattern. To do so, we propose a simple, yet powerful, extension to regular expressions, called `TreeRegex`, to specify patterns over serialized trees. We next introduce `TreeRegex` gradually through a series of simple examples to demonstrate its intuitiveness.

In our examples, we assume that the inputs are strings denoting arithmetic expressions constructed using positive integers and arithmetic operators $+, -, *, /, (,)$. We assume that the strings have no space or newline characters. We also assume that the strings have been parsed and converted into serialized trees using a parser with standard precedence declaration for arithmetic operators. For simplicity of exposition and to reduce clutter, we assume that integer literals are not enclosed within ($(^{\%}$ and $_{\%})$). For example, the arithmetic expression string "$3 * 4 + 5 * (6 - 2)$" has been converted to the serialized tree "$(^{\%}(^{\%}3 * 4_{\%}) + (^{\%}5 * (^{\%}((^{\%}6 - 2_{\%}))_{\%})_{\%})_{\%})$".

Matching an exact serialized tree. Let us first write a pattern that checks if an arithmetic expression is the addition of two positive integers. For example, "2 + 3 + 1" does not match this pattern, but "2 + 3" matches the pattern. Such a pattern can be easily written using the regular expression: `\d+\+\d+`. Here `\d` denotes the digit character class and `\d+` denotes one or more digits. Since `+` is a meta-character in regular expressions, we escape `+` with `\` to denote the actual `+` arithmetic operator. When matching against the serialized tree corresponding to an arithmetic expression, we need to use a `TreeRegex` expression. In `TreeRegex`, we extend regular expressions by allowing the usage of parenthesis meta-characters ($(^{\%}$ and $_{\%})$) in a balanced fashion. For example,

$$(^{\%}\backslash d+\backslash +\backslash d+_{\%})$$

is a `TreeRegex` expression and it matches the serialized tree (e.g. "$(^{\%}2 + 3_{\%})$") corresponding to an arithmetic expression where two positive integers are being added. It is important to note that a regular expression in a `TreeRegex` expression cannot match the meta-characters ($(^{\%}$ and $_{\%})$). Another example of a `TreeRegex` expression that matches an arithmetic expression is one that denotes the addition of two expressions, where each expression is the multiplication of two positive integers. This expression is: $(^{\%}(^{\%}\backslash d+\backslash *\backslash d+_{\%})\backslash +(^{\%}\backslash d+\backslash *\backslash d+_{\%})_{\%})$. This `TreeRegex` expression matches the serialized tree "$(^{\%}(^{\%}31*4_{\%})+(^{\%}5*62_{\%})_{\%})$" (i.e. serialized tree for "$31 * 4 + 5 * 62$").

Matching an arbitrary serialized tree. So far, we have extended regular expressions with parenthesis meta-characters ($(^{\%}$ and $_{\%})$)—a `TreeRegex` expression with this extension has the form of a serialized tree. However, this extension is not enough if we want to match more complex patterns. For example, suppose we want to write a pattern that matches an arithmetic expression that is the addition of *two arbitrary arithmetic expressions*. We need a (sub-)`TreeRegex` expression that matches an arbitrary arithmetic expression. In the general case, we want a pattern that matches an arbitrary serialized tree beginning and ending with ($(^{\%}$ and $_{\%})$), respectively. We add the meta-character **@** to `TreeRegex` to match such arbitrary serialized trees. A `TreeRegex` expression that matches the addition of two arbitrary arithmetic expressions could then be written as

$$(^{\%}@\backslash +@_{\%}).$$

Here **@** matches any serialized tree that starts with a $(^{\%}$ and ends with a $_{\%})$. Note that **@** cannot match any arbitrary string. This `TreeRegex` expression will now match "$(^{\%}(^{\%}31 * 4_{\%}) + (^{\%}5 * 62_{\%})_{\%})$" (i.e. serialized tree for "$31*4+5*62$"), and "$(^{\%}(^{\%}2+3_{\%})+(^{\%}1*4_{\%})_{\%})$" (i.e. serialized tree for "$2 + 3 + 1 * 4$"). It will not match "$(^{\%}2 + 3_{\%})$", because "2" and "3" do not start with $(^{\%}$ and end with $_{\%})$. Note that we did not enclose an integer literal in ($(^{\%}$ and $_{\%})$) to illustrate this subtlety; our implementation does enclose integers in ($(^{\%}$ and $_{\%})$).

Matching a serialized tree nested in another serialized tree. Now suppose we want to match any arithmetic expression that contains a specific form of nested sub-expression. The form we are looking for is an addition of two integers, and it can be nested arbitrarily deep inside the top-level expression. For example, the pattern should match both "$(^{\%}(^{\%}2 * (^{\%}((^{\%}3 + 11_{\%}))_{\%})_{\%}) * 1_{\%})$" (i.e. serialized tree for "$2*(3+11)*1$") and "$(^{\%}2+3_{\%})$ (i.e. serialized tree for "$2+3$"). We now need the ability to specify a pattern that matches a serialized tree that contains a serialized tree at an arbitrary depth. To do so, we add two more parentheses meta-characters ($(^{*}$ and $_{*})$) (inspired by Kleene star in regular expressions) to `TreeRegex`. A `TreeRegex` expression can use these meta-characters as long as the expression is balanced with respect to both ($(^{\%}$, $_{\%})$), and ($(^{*}$, $_{*})$). A pattern $(^{*}\texttt{t}_{*})$, where `t` is some other `TreeRegex` expression, matches any serialized tree that contains a nested serialized tree matching `t`. With this new extension, the `TreeRegex` expression

$$(^{*}\backslash d+\backslash +\backslash d+_{*})$$

matches an arithmetic expression that has a nested arithmetic sub-expression that is the addition of two positive integers.

Revisiting the motivating example. We are now ready to write a `TreeRegex` expression that checks if a JavaScript program contains an `eval`-calling function body. First note that a function definition in a JavaScript program can be arbitrarily nested inside the program. A call to `eval` can be arbitrarily nested within the body of a function as well. Therefore, we need two sets of ($(^{*}$, $_{*})$): one pair to match a function definition and another pair to match a call to `eval`. The `TreeRegex` expression for the pattern is

`(*function .*(@){(*eval(@)*)}*)`

This pattern matches the serialized tree of a JavaScript program if the program has a function whose body calls `eval`. The first @ matches the serialized tree for the list of parameters, and the second @ matches the serialized tree for the argument to `eval`. Note that while `.*` matches an arbitrary string, it cannot match a serialized tree—if the name of function included structured information, like a template type in C++, a `TreeRegex` matching a serialized tree would be required. Similarly, @ matches an arbitrary serialized tree, but it cannot match any string.

2.2 Capture Group and Replacement

Most regular expression libraries provide support for search-and-replace via replacement strings. We provide support for similar search-and-replace operations in `TreeRegex`.

Let us consider the motivating example again. Now we want to replace the `eval` call with a `safe_eval` call. To do this, we make slight modifications on the `TreeRegex` expression as follows:

`(%function ⦅.*⦆ (@){(*eval(@)*)}%)`

We simplify the example by replacing the outermost `*`-parentheses with `%`-parentheses, and add ⦅ and ⦆ to capture the name of the function. We use ⦅ and ⦆ to denote regular expression parenthesis metacharacters that capture. Now we need to capture four pieces of information: the name of the function, the formal parameters, the argument of the `eval` function call, and the text that surrounds the `eval` function call. The function name is matched and captured by the ⦅`.*`⦆. In `TreeRegex`, we specify that the wildcard @ captures the serialized tree it matches. This means the formal parameters and the argument passed to the `eval` function are captured. We can now build our desired replacement string

`(%function $1($2){. . . safe_eval($4). . . }%)`

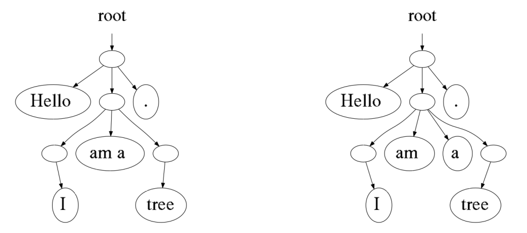
where `$1` and `$2` refers to the first and second captured values, `$4` refers to the value captured by @ (which is the argument passed to the `eval` function), and `. . .` are the missing strings that we are yet to specify.

The strings that surround the `eval` function call are more difficult to manipulate because there is no replacement string syntax in conventional regular expressions for inserting a string in the middle of a captured value.

We need to insert our new `safe_eval` function call between the strings that are to the left and right of the `eval` function call. Before that, though, we need to capture the strings to the left and right of the `eval` function call. In `TreeRegex`, we specify that an expression of the form `(* t *)` creates a capture group that captures a string with a *hole*. For example, if `(*eval(@)*)` matches the string `bar(); foo(eval(s),2);` , then the `(*`, `*)` pair will capture the string `bar(); foo(•,2);` , which has a hole •. This captured string will be referred by `$1` in this case. With this new definition of a capture group, we can specify our desired replacement string as

`(%function $1($2){$3(%safe_eval($4)%)}%)`

In this replacement string, `$3` represents the strings surrounding the `eval` function call. This string has a hole. The question is what



(a) A valid tree. **(b) An invalid tree.**

Figure 1: Example trees. The first tree is equivalent to serialized tree “(%Hello (%(%I%)am a(%tree%)%).%)”. The second tree is invalid, and doesn't have a corresponding serialized tree.

string do we use to fill the hole. In our approach, we fill the hole with the serialized tree that follows `$3` in the replacement string, which in our case is the string `(%safe_eval($4)%)` where `$4` is suitably replaced.

In summary, in `TreeRegex` a @ captures a serialized tree string, and `(*t*)` captures a string with a hole. If in a replacement string, `$n` refers to a string with hole, then the hole is filled with the serialized tree string that follows `$n` in the replacement string.

3 FORMAL DESCRIPTION

We formalize the behavior of `TreeRegex` expressions and replacements in this section. We begin with describing serialized trees and how to construct them from a tree data-structure in 3.1. We then describe the syntax, semantics, and a matching algorithm for `TreeRegex` expressions in 3.2, 3.3, and 3.4, respectively. We describe the replacement algorithm in 3.5. Finally, the time/space complexity of the matching algorithm is discussed in 3.6.

3.1 Serialized Tree

A `TreeRegex` expression is a pattern that matches against a serialized tree. A serialized tree is formed from a *tree*. A tree is recursively defined as follows. A tree is a non-empty list each element of which is either a tree or a non-empty string. Moreover, two strings in a list cannot be next to each other. Figure 1a shows a valid tree. The tree in Figure 1b is not a valid tree because there are two consecutive nodes in the tree that are strings.

For the purpose of `TreeRegex` matching, we assume that an input tree against which we need to match a `TreeRegex` expression is given in a serialized form as a string. This serialized tree form, which resembles *S-Expressions* in LISP [53], can be constructed recursively from a tree as follows. The serialized tree of a string is the string itself. For a list, compute the serialized trees of its elements: the serialized tree of a tree is computed recursively. Once we have the serialized trees of the elements of the list, we concatenate them and surround the resulting string with (% and %). This gives us the serialized tree of the tree represented by the list.

For example, the serialized tree for the tree in Figure 1a is “(%Hello (%(%I%)am a(%tree%)%).%)”. Note that a serialized tree retains the structure of the tree by surrounding each sub-tree with (% and %). If we remove occurrences of (% and %) from a serialized tree, we get back the original string.

We need a formal grammar for serialized trees in order to describe the matching and replace algorithms for `TreeRegex`. We use the following grammar to describe the set of all serialized trees.

$\langle \mathit{Tree} \rangle$ ::= $({}^{\%}\ \langle \mathit{ListOfTrees} \rangle\ {}_{\%})$

$\langle \mathit{ListOfTrees} \rangle$ ::= $(\langle \mathit{String} \rangle ? \langle \mathit{Tree} \rangle)^{*} \langle \mathit{String} \rangle ?$

$\langle \mathit{String} \rangle$ ::= $u \in \Sigma^{+}$

In a grammar, we surround non-terminals with $\langle\rangle$. $\langle \mathit{Tree} \rangle$ denotes the root of a tree. This non-terminal has a starting $({}^{\%}$ to distinguish it from a leaf, a list whose each element is either a tree or a string (denoted by $\langle \mathit{ListOfTrees} \rangle$), and then a closing ${}_{\%})$ to end the list. The production rule for $\langle \mathit{ListOfTrees} \rangle$ denotes a non-empty list of alternating strings and sub-trees. For simplicity, we use a regular expression to describe the production rule as in Extended Backus-Naur Form [37]. The rule ensures that any two strings are non-consecutive in the list. This property is important to prevent ambiguity in transforming an input to the corresponding serialized tree, as observed in the example of Figure 1b. $\langle \mathit{String} \rangle$ is the grammar for strings: they are of non-zero length, from the alphabet Σ. *We assume Σ does not contain the metacharacters $({}^{\%}$ and ${}_{\%})$, or the* `TreeRegex` *metacharacters* **@**, $({}^{*}$, *and* ${}_{*})$. [5] In the rest of the paper, we will use the symbols s, s', s_1, s_i, s_n etc. to denote serialized trees or strings over Σ. We use Σ_s to denote the alphabet that contains Σ plus all the meta-characters and a serialized tree is a string over Σ_s.

3.2 TreeRegex

`TreeRegex` is a simple extension to regular expressions. The extension has been designed keeping in mind that we want to describe patterns not only over simple linear strings, but also on serialized trees. The following is the grammar for `TreeRegex`.

$\langle \mathit{TreeRegex} \rangle$ ::= $({}^{\%} \langle \mathit{ListOfTreeRegexes} \rangle\ {}_{\%}) \mid ({}^{*} \langle \mathit{ListOfTreeRegexes} \rangle\ {}_{*}) \mid$ **@**

$\langle \mathit{ListOfTreeRegexes} \rangle$ ::= $(\langle \mathit{Regex} \rangle ? \langle \mathit{TreeRegex} \rangle)^{*} \langle \mathit{Regex} \rangle ?$

$\langle \mathit{Regex} \rangle$ ::= a regular expression over Σ

The grammar is similar to that of a serialized tree. A `TreeRegex` expression, denoted by $\langle \mathit{TreeRegex} \rangle$, can be of three types: exact expressions, context expressions, and wildcard expressions. An exact expression starts with a $({}^{\%}$ followed by a list whose each element is either a `TreeRegex` expression or a regular expression, and finishes with a ${}_{\%})$. The list is denoted by $\langle \mathit{ListOfTreeRegexes} \rangle$ and cannot have two regular expressions next to each other. This restriction naturally follows from the similar restriction posed on serialized tree. A context expression is visually similar to an exact expression, except that it starts with a $({}^{*}$ and ends with a ${}_{*})$, using symbols inspired by the Kleene star[6]. A wildcard expression is denoted by the terminal symbol **@** (and it contains no $\langle \mathit{ListOfTreeRegexes} \rangle$ expressions). A regular expression describes a regular language over the alphabet Σ which does not contain $({}^{\%}, {}_{\%})$, $({}^{*}, {}_{*})$, and **@**. *Thus a regular expression (including* `.*`*) cannot match any of these meta-characters.* We will use the symbols t, t', t_1, t_i, t_n etc. to denote `TreeRegex` expressions.

[5]In the actual implementation we escape the meta-characters in a string suitably.

[6]It operates like a Kleene star, except instead of matching multiple characters on a single level of a tree, it matches many levels of a tree.

3.3 Language of TreeRegex

A `TreeRegex` expression t describes a set of serialized trees, denoted by $L(t)$. $L(t)$ is defined recursively as follows.

Exact Expressions. If t is a an exact expression of the form $({}^{\%}t_1 \ldots t_{n\%})$, then the language of this expression is: $L(({}^{\%}t_1 \ldots t_{n\%})) = \{({}^{\%}s_1 \ldots s_{n\%}) \mid s_1 \in L(t_1) \ldots s_n \in L(t_n)\}$. For a string to be in the language of $({}^{\%}t_1 \ldots t_{n\%})$, it must be constructed from an element of the language of each t_i and then surrounded by $({}^{\%}$ and ${}_{\%})$. That is, if each s_i is in $L(t_i)$ then $({}^{\%}s_1 \ldots s_{n\%}) \in L(({}^{\%}t_1 \ldots t_{n\%}))$.

Context Expressions. If t is a context expression of the form $({}^{*}t_1 \ldots t_{n*})$, then it describes the language where each string is a serialized tree containing a string from $L(({}^{\%}t_1 \ldots t_{n\%}))$ as some subtree. In order to define the language of a context expression formally, we need to define a serialized tree context. A *serialized tree context* is a serialized tree where some subtree is replaced by a hole. The subtree does not need to be immediate—it can be the subtree of a subtree that is the hole, for instance. The hole is denoted by a $\bullet$. Given a serialized tree context c and a serialized tree s, we use $c(s)$ to denote the serialized tree obtained by replacing the hole in c by s. Then we can define $L(({}^{*}t_1 \ldots t_{n*})) = \{c(s) \mid s \in L(({}^{\%}t_1 \ldots t_{n\%}))$ and c is any serialized tree context$\}$

Wildcard Expressions. If t is a wildcard expression (i.e. if $t =$ **@**), then $L(t)$ is the set of all serialized trees.

Regex Expressions. We do not define the language of regular expressions since it is a well-studied topic and for the purpose of defining `TreeRegex`, we do not need a formal definition of regular expressions. We just assume that the language of a regular expression is a subset of the strings in Σ^{*}, where Σ is our string alphabet.

3.4 TreeRegex Matching Algorithm

Next we describe an algorithm that matches a string against a `TreeRegex` expression. We say that a `TreeRegex` expression t matches a serialized tree s if $s \in L(t)$. Similar to conventional regular expressions, `TreeRegex` allows us to not only match against a string, but also to extract strings, serialized trees, and serialized tree contexts for further processing. In conventional regular expressions, this is achieved by defining groups of characters and capturing them using the parenthesis capture group meta-characters. A string that matches a nested/sub-regular expression within a pair of parentheses gets captured as a group. In the case of `TreeRegex`, a serialized tree that matches a wildcard expression or a serialized tree context that matches a context expression gets captured as a group. Note that in case of `TreeRegex` we do not need to explicitly use parentheses to define a capture group—any wildcard expression or context expression implicitly defines a capture group.

We describe a function *match* which takes a `TreeRegex` expression t and a serialized tree s. It returns a list of *captures* if s matches t and returns *nil*, which we distinguish from an empty list, otherwise. We will denote lists of captures using the symbols K, K', K_1, K_i, K_n etc. The function *match(t, s)* is defined recursively as follows:

- **Case 1.** t is a regular expression and s is a string in Σ^{+} and not a serialized tree: If t matches the string s using conventional regular expression matching algorithm, then *match(t, s)* returns a list of *captures* that one gets from the regular expression matching algorithm.

- **Case 2.** t is of the form $({}^{\%}t_1 \ldots t_{n\%})$ and s is of the form $({}^{\%}s_1 \ldots s_{m\%})$: If $n = m$ and *match(t_1, s_1)*, …, *match(t_n, s_n)* returns the lists $K_1, \ldots, K_n$, respectively, then *match(t, s)* returns the list $K_1 \cdot K_2 \ldots K_n$ if none of K_i's are *nil*. (We use $K_1 \cdot K_2$ to denote the list obtained by concatenating lists K_1 and K_2.)
- **Case 3.** t is **@** and s is of the form $({}^{\%}s_1 \ldots s_{m\%})$: *match(t, s)* returns $[s]$. (We use $[s]$ to denote the list containing a single element s.)
- **Case 4.** t is of the form $({}^{*}t_1 \ldots t_{n*})$ and s is of the form $({}^{\%}s_1 \ldots s_{m\%})$: we consider the following two cases, the second of which is recursive. Note that in the recursive step, the entire depth of the tree may be matched against.
  - If *match(($^{\%}t_1 \ldots t_{n\%}$), s)* returns a list, say K, then *match(t, s)* returns the list obtained by prepending the serialized tree context • to K, i.e. returns the list $[\bullet] \cdot K$.
  - Otherwise, if there exists a i such that *match(t, s_i)* returns a list of the form $[e] \cdot K$, then *match(t, s)* returns the list $[({}^{\%}s_1, \ldots, s_{i-1}, e, s_{i+1}, \ldots, s_{n\%})] \cdot K$. If multiple such i's exist, the first is chosen.
- **Default Case.** t and s do not match any of the above cases: *match(t, s)* returns *nil*, which we use as a "bottom" value and is not the same as an empty list.

A formal description of the function *match(t, s)* can be found in Appendix B.

3.5 `TreeRegex` Replacement Algorithm

Most regular expression libraries provide support for search-and-replace via replacement strings. A replacement string is the string that a regular expression match is replaced with during a search-and-replace operation. Replacement strings usually are strings with special meta-characters of the form $\$n$, where n is a positive integer. During a replacement action each $\$n$ in the string gets replaced by the n^{th} *capture* while matching the regular expression against a string. In `TreeRegex`, we support similar search-and-replace capabilities. A replacement string in `TreeRegex` is a serialized tree which could contain special meta-characters of the form $\$n$, where n is a positive integer. These special meta-characters will be replaced by strings, serialized trees, or serialized tree contexts. The replacement algorithm works in a straight-forward way for *captures* in the form of strings and serialized trees: we simply replace a $\$n$ in the replacement string with the n^{th} *capture* during `TreeRegex` matching. However, the algorithm gets slightly complicated when we have a *capture* in the form of a serialized tree context.

We now define the function *replace(r, K)*, which takes a replacement string r and a list K of *captures* captured during a `TreeRegex` matching and returns a new serialized tree. We assume that all meta-characters $\$n$ appearing in the replacement string have a corresponding item in the list of *captures*. We use $K(i)$ to denote the i^{th} *capture* in K, w_1 and w_2 to denote arbitrary strings over Σ_s (i.e. strings containing meta-characters and characters from Σ), s to denote a serialized tree, c to denote a serialized tree context. In the *replace* function, we replace each $\$n$ with a string or a serialized tree as follows.

- **Case 1.** r is the string $w_1\$nw_2$ (where w_1 and w_2 are arbitrary strings) and $K(n)$ is a string: Function *replace* replaces $w_1\$nw_2$ with the string obtained by concatenating w_1, $K(n)$, and w_2.
- **Case 2.** r is the string $w_1\$nw_2$ and $K(n)$ is a serialized tree: As before, function *replace* replaces $w_1\$nw_2$ with a string obtained by concatenating w_1, $K(n)$, and w_2. Note that $K(n)$ is a serialized tree, so it is a string in Σ_s.
- **Case 3.** r is the string $w_1\$nsw_2$ and $K(n)$ is a serialized tree context. If $K(n)$ is a serialized tree context, then we do a replacement only if a serialized tree, s, follows $\$n$ in r; we use s to fill up the hole in $K(n)$. Function *replace* first replaces the hole • in $K(n)$ with s to get the serialized tree[7] $K(n)(s)$, then replaces $w_1\$nsw_2$ with a string obtained by concatenating w_1, $K(n)(s)$, and w_2.

We apply the above steps repeatedly until no more replacements can be performed. Note that in the third case above, if $K(n)$ is a serialized tree context and $\$n$ is not followed by a serialized tree in r, we skip the replacement of $\$n$ until *replace* converts r into a r' where $\$n$ is followed by a serialized tree. Because of this, after the termination of the algorithm we may end up in a serialized tree which contains a meta-character of the form $\$n$. In that case, replacement has failed and we raise an exception. For example, the replacement string $({}^{\%}a\$1c_{\%})$ will fail if the list $K = [\bullet, ({}^{\%}b_{\%})]$. However, when the same list is used with the replacement string $({}^{\%}a\$1\$2c_{\%})$, the algorithm first replaces $\$2$ with $({}^{\%}b_{\%})$ to yield $({}^{\%}a\$1({}^{\%}b_{\%})_{\%})$ [second case] and then fills the hole in $\$1$ with $({}^{\%}b_{\%})$, yielding the final string $({}^{\%}a({}^{\%}b_{\%})c_{\%})$ [third case]. A formal description of the algorithm can be found in Appendix C.

3.6 Running Time Complexity

The time complexity of the matching algorithm is bounded by $O((mn)^{k+1})$, where m is the size of the `TreeRegex` expression, n is the size of the serialized tree, and k is the number of context expressions in the `TreeRegex` expression. The algorithm has a space complexity of $O(k)$. One can also use memoization during matching to come up with an algorithm whose time complexity is $O(mn)$ and space complexity is $O(mn)$. In our implementation we do not use memoization because k is usually 1 or 2 in our usage. A detailed complexity analysis of the algorithm can be found in Appendix D. The replacement algorithm is straight-forward and has a time complexity of $O(mn)$, where where m is the size of the replacement string and n is the size of the serialized tree.

4 CONSTRUCTING `TREEREGEX` EXPRESSIONS

To perform matching or replacement in a tree-structured text, we need to construct suitable `TreeRegex` expressions. This could become tedious if we need to construct large number of `TreeRegex` expressions from scratch. In our case studies, we found that if we take a look at a couple of source and target serialized trees, we can easily write our desired `TreeRegex` expressions and replacement strings. We next describe a simple process that we used to derive `TreeRegex` expressions and replacement strings from examples. The process significantly helped us through our case studies. The process has three steps: creating a few examples of serialized trees

[7]Note that if c is a serialized tree context and s is a serialized tree, then $c(s)$ is the serialized tree obtained by replacing the hole in c with s.

before and after the transformation, stripping out irrelevant context and details via “diff”-ing, and identifying replacement indices. We use a simple example to demonstrate the process: finding and instrumenting the conditions of if-statements in JavaScript.

Collecting Relevant Serialized Trees. In order to derive a `TreeRegex` expression for a particular task, we first create a few sample programs before and after transformation. To search for if-statements in JavaScript programs, we use the following example input programs:

- `if(x<0) m--;`
- `function f(){if(k==3){i*4;}}`

We specifically varied the condition and the body of the if-statements so that the only commonality between the samples is the presence of an if-statement. The desired instrumented forms are as follows:

- `if(Cond(x<0)) m--;`
- `function f(){if(Cond(k==3)){i*4;}}`

Typically only two to three examples are necessary to derive the correct `TreeRegex` expression. Examples with comments or unusual whitespace are also useful so that the user can determine if the parser removes them or treats them as separate sub-serialized trees. For the duration of this section, we will consider a parser that removes these artifacts.

We next convert these example programs into the following serialized trees:

- $(^{\%}\mathtt{if}((^{\%}(^{\%}\mathtt{x}_{\%})<(^{\%}\mathtt{0}_{\%})_{\%}))(^{\%}(^{\%}(^{\%}\mathtt{m}_{\%})\mathtt{--}_{\%});_{\%})_{\%})$
- $(^{\%}\mathtt{function\ f()}(^{\%}\ \{(^{\%}\mathtt{if}((^{\%}(^{\%}\mathtt{k}_{\%})==(^{\%}\mathtt{3}_{\%})_{\%}))(^{\%}\ \{(^{\%}(^{\%}(^{\%}\mathtt{i}_{\%})*(^{\%}\mathtt{4}_{\%})_{\%});_{\%})\}_{\%})_{\%})\}_{\%})_{\%})$

We generate similar serialized trees for the instrumented JavaScript programs.

Stripping Out Irrelevant Context and Details via Diff-ing. Each of the serialized trees, obtained from the examples, have additional context and details that are unimportant to the task of instrumentation. For instance, the body of each if-statement does not change the instrumentation behavior—this is an irrelevant *detail.* Whether an if-statement is in a function or in the global scope is equally unimportant—this is an irrelevant *context.*

To detect these irrelevant context and details, we perform a “diff” on each set of serialized trees. The “diff” divides each serialized tree into those parts that are in common with each other serialized tree in the set, and those parts which are not in common. Following is the diff result of the set of serialized trees obtained from the two example input programs:

- $(^{\%}\mathtt{if(}\ (^{\%}(^{\%}\mathtt{x}_{\%})<(^{\%}\mathtt{0}_{\%})_{\%})\ )\ (^{\%}(^{\%}(^{\%}\mathtt{m}_{\%})\mathtt{--}_{\%});_{\%})\ _{\%})$
- $(^{\%}\mathtt{function\ f()}(^{\%}\ \{\ (^{\%}\mathtt{if(}\ (^{\%}(^{\%}\mathtt{k}_{\%})==(^{\%}\mathtt{3}_{\%})_{\%})\ )$
  $(^{\%}\ \{(^{\%}(^{\%}(^{\%}\mathtt{i}_{\%})*(^{\%}\mathtt{4}_{\%})_{\%});_{\%})\}_{\%})\ _{\%})\ \}_{\%})_{\%})$

The highlighted parts represent the differences of two serialized trees. They are either the irrelevant context or the irrelevant details. If the highlighted part is outside of the non-highlighted parts, then it is the irrelevant context and can be removed. If it is inside, then it is an irrelevant detail. For each irrelevant detail that is a serialized tree, we replace it with an indexed wildcard `TreeRegex` expression ($@_i$). For each irrelevant detail that is a string, we replace it with an indexed ⦅.*⦆$_i$ regular expression. This distinction is necessary because **@** expressions do not match non-serialized trees and regular expressions do not match serialized trees. The following are the results after this step. The left items are obtained from the example input programs, and the right items are from the example instrumented programs:

- $(^{\%}\mathtt{if}(@_1)@_{2\%})$
- $(^{\%}\mathtt{if}((^{\%}(^{\%}\mathtt{Cond}_{\%})(@_5)_{\%}))@_{6\%})$
- $(^{\%}\mathtt{if}(@_3)@_{4\%})$
- $(^{\%}\mathtt{if}((^{\%}(^{\%}\mathtt{Cond}_{\%})(@_7)_{\%}))@_{8\%})$

During the process, we maintain *a mapping from the indexed expressions to the concrete serialized trees and strings they have replaced.* For example, $@_1$ and $@_5$ refer to $(^{\%}(^{\%}\mathtt{x}_{\%})<(^{\%}\mathtt{0}_{\%})_{\%})$, and $@_2$ and $@_6$ refer to $(^{\%}(^{\%}(^{\%}\mathtt{m}_{\%})\mathtt{--}_{\%})_{\%})$. This mapping will be used in the next step.

Assigning Replacement Indices. The last step is to construct replacement strings for the `TreeRegex` expressions obtained in the previous step. To do so, we first match each $@_i$ from the pre-instrumentation `TreeRegex` expression with $@_j$ from the post-instrumentation `TreeRegex` expression such that they map to the same serialized tree (e.g. $@_1$ and $@_5$). Thus to obtain the replacement string corresponding to the post-instrumentation version, we replace $@_5$ with `$1`, where 1 is the index of the capture group corresponding to $@_1$ in the pre-instrumentation version.

We construct the final `TreeRegex` expressions by dropping the indices from the **@** expressions in the `TreeRegex` expressions in the pre-instrumentation set. Additionally, one could further constrain the ⦅.*⦆ regular expressions with more constrained regular expressions, e.g. with ⦅\d+⦆. Lastly, we remove any duplicate pattern pairs, yielding the following `TreeRegex` expression and replacement expression:

- $(^{\%}\mathtt{if}(@)(^{\%}\{@\}_{\%})_{\%})$
- $(^{\%}\mathtt{if}((^{\%}(^{\%}\mathtt{Cond}_{\%})(\$1)_{\%}))(^{\%}\{\$2\}_{\%})_{\%})$

5 TRANSFORMERS AND IMPLEMENTATION

We have implemented `TreeRegexLib`, a match-and-replace engine for `TreeRegex`, in C++ (926 lines), Java (1044 lines), and JavaScript (746 lines). We use the existing, unmodified regular expression libraries of these languages for matching regular expressions. We believe that if a language has a library for regular expressions, it is straight-forward to implement `TreeRegexLib`. In our implementations, we use *a simple tree data-structure to denote a serialized tree.* This helps us to avoid unnecessary serialization and parsing of a serialized tree while performing multiple `TreeRegex` matching and replacements.

Both implementation provide the `transformer` API as a primary interface to search and manipulate an AST. `TreeRegex` expressions and replacement expressions are good at describing a single match-and-replacement task. However, an AST-manipulating program often needs more than that. For example, it may need to find all subtree matching a given pattern, to accumulate information from each matching subtree, and to perform different actions based on the collected information rather then just depending on the syntactic pattern. The `transformer` API is designed to support such tasks, using `TreeRegex` expressions and replacement expressions as components.

With the `transformer` API, an AST manipulation task can be described using a collection of `transformers`. A `transformer` is defined as a tuple of the form (type, t, M, r), where type is either *pre* or *post*, t is a `TreeRegex` expression, M is the modifier, which is a function taking a list of *captures* and a user-defined state, and returning a possibly modified list of *captures*, and *r* is a replacement string. Each transformer essentially describes a single match-and-replacement task, with additional components (a *pre*/*post* tag and a modifier). Having a collection of `transformers`, the `transformer` API traverses an input serialized tree in depth-first manner, applying all transformers in the collection to each sub-serialized tree. When a `transformer` is applied to a sub-serialized tree, the following actions take place.

Only `transformers` of type *pre* are applied to a sub-serialized tree before its children have been visited, and only `transformers` of type *post* are applied to a sub-serialized tree after its children have been visited. While visiting a sub-serialized tree, *t* is matched against the sub-serialized tree to obtain a *capture* list, say K.

Next modifier M of the `transformer` is applied to the list of *captures* K and the user-defined state σ. The modifier could change the user-defined state σ and return a potentially modified list of *captures*. A default implementation of a modifier returns the list of *captures* passed as argument. The modifier is also a suitable place where a warning could be printed or where necessary information could be accumulated to the user-defined state. If the modifier returns a non-*nil* list of captures, a new serialized tree is created from the replacement string *r* using the list of *captures*. The new serialized tree replaces the current sub-serialized tree. If a replacement string is not provided, the current sub-serialized tree is kept unmodified. A formal description of the transformation algorithm can be found in Appendix E.1 and an example usage can be found in Appendix E.2.

6 CASE STUDIES

We report on five case studies where we found that `TreeRegexLib` significantly simplifies various tasks related to manipulating tree-structured texts. Note that a `TreeRegex` expression depends on the structure of the serialized tree of the target language. One may think that this could pose a problem if we change our serialization format frequently. We usually create the most generic serialized tree for a given AST, where each internal node of the AST is enclosed with ($^{\%}$ and $_{\%}$). Such a format does not change unless we switch to a different parser and AST. In general, once we fix a parser for a language, the serialized tree format for the language is fixed as well. For example, we used the same serialization format for both of our JavaScript case studies. We did not find any need to change the serialization format from one application to another.

6.1 Measuring JavaScript Test Coverage

In this case study, we use `TreeRegexLib` to instrument JavaScript programs for tracking branch and statement coverage. The instrumentor has two parts: a JavaScript program to serialized tree *converter*, and a list of `transformers` implementing the instrumentation. We built our converter on top of the `acorn` parser [6] and `esotope` [10] JavaScript code generator by adding 108 lines of modification.

Our instrumentation program wraps the conditional expressions in various statements and expressions, such as `if-else`, `for`, `while`, and `switch`. For example, the code `if (x>0){x = 0;}` gets instrumented into `if (Cond(id, x>0)){x = 0;}`. A unique static `id` is passed as the first argument to `Cond` and the conditional expression is passed as the second argument. An implementation of `Cond` records the branch being taken and returns the value of the conditional expression unmodified. Similarly, we add a call to `Stmt` before every statement to track statement coverage.

The instrumentation program has 13 `transformers` implemented in 37 lines of JavaScript code. A simplified version of the `TreeRegex` expression and replacement string used to instrument an `if`-statement is shown below.

```
(%if (@) @ %)        (%if (Cond($3, $1)) $2%)
```

Here $3 represents a static id which is generated and appended to the list of captures in the modifier of the `transformer`.

The instrumentation program was quite straight-forward to write. The total lines of code of the instrumentation program, which is 145 including the serialized tree converter, is significantly fewer than the 968 lines of code of the instrumentor in *istanbul* [13], a popular JavaScript coverage tool. Istanbul uses a similar parser, called `esprima` [11], and the `esotope` code generator. It programmatically visits the AST of a JavaScript program to perform the instrumentation. We believe that such traversal code is tedious to write, debug, and maintain. Another important aspect of our instrumentation tool is that we did not use a specialized term-rewriting tool to perform instrumentation. Such a tool would simplify the task of writing an instrumentor; however, it would require one to define a grammar for JavaScript. We simply reused an existing parser and code generator. The ability to exploit existing tools for generation of serialized trees makes `TreeRegexLib` practical for real-world usage.

6.2 Detecting Injection Attacks

In this case study, we show that `TreeRegex` can be used to detect injection attacks. Injection is a class of attacks that works by injecting data into a program template (i.e. a program with missing portions to be filled with data) in order to facilitate the execution of a malicious program that alters the intended behavior of the original program. Examples of injection attacks include cross-site scripting (XSS), SQL injection, injection in strings passed to JavaScript's `eval` function and `system` C function, shell variable expansions.

We focused on SQL injection attacks in this case study because they are quite common: in the past 4 years, there are over 700 CVE reports of distinct SQL injection vulnerabilities [1]. Therefore, a significant corpus of vulnerabilities is available for evaluation. Since `TreeRegexLib` is not specifically designed for SQL, we expect that our technique is portable to other types of injection attacks.

There is a significant volume of research on detecting injection attacks in SQL [17, 30, 54, 64, 65]. The most popular techniques to detect SQL injection attacks ensure that the injected string is not treated as instructions but as a string value. Instead of checking if the injected string has a restricted format, we use `TreeRegex` expressions to check if, after injection, there is any alteration in the syntactic structure of the resultant string. We believe that our approach to check the resultant string instead of checking the injected string is quite powerful. In most languages, this is sufficient to determine whether an attack has occurred.

We evaluate our technique on existing SQL injection vulnerabilities in Wordpress [5] plugins. Wordpress plugins are ideal for evaluating injection-detection techniques because (1) Wordpress does not require plugins to use a safe SQL command API, (2) Wordpress has a large installation base [5], and (3) vulnerable plugins are easily available. From an exploit database [12] we downloaded the last 4 years of vulnerable Worldpress plugins with exploit code. This amounted to 32 plugins, 390k lines of PHP source code, and approximately 2000 SQL commands. For each exploit, we read the exploit writeup from the database and found the vulnerable SQL command. We used the technique from Section 4 to construct `TreeRegex` expressions that matched non-exploited versions of the commands[8]. We used the constructed `TreeRegex` expressions to detect any exploited vulnerability at runtime by inserting a check before the SQL command is evaluated.

We constructed 34 `TreeRegex` expressions, with an average length of 61 tokens, for the 32 vulnerable plugins. We were able to detect all exploits, with neither false-positives nor false-negatives. All of the vulnerable SQL commands could be parsed into serialized trees. To understand how the expressions were able to detect injection attacks, consider these two SQL serialized trees from function calls of the *cp-multi-view-calendar* plugin, version 1.1.7 [8]:

```
(%update (%‘wp_dc_mv_events’%) set
   (%‘exdate’%)=(%‘’%) where (%(%‘id’%)=(%4%)%)%)
(%update (%‘wp_dc_mv_events’%) set (%‘exdate’%)=(%‘’%)
   where (%(%‘id’%)=(%(%SLEEP%)((%2%))%)%)%)
```

In the second serialized tree, the `SLEEP` SQL function is an injected behavior change. The intended structure is straightforward: an update SQL statement sets the variable `exdata` to be an empty string where `id` has a certain value. We can create a `TreeRegex` expression specifying the expected structure of the SQL command:

```
(%update (%.*%) set (%‘exdata’%)=(%.*%)
   where (%(%‘id’%)=(%.*%)%)%)
```

Note that we are using the regular expression `.*` to prevent serialized trees from appearing in certain positions above. The regular expression `.*` only matches strings and not serialized trees—therefore, any attempt to inject a syntactic structure, which is not a string, would be blocked. In the example, we are restricting the right-hand side of the assignments to `extdata` and `id`, along with the table name.

6.3 Linter for JavaScript

In this case study, we re-implemented a subset of the code-checking rules used in the popular JavaScript linting tool, `ESLint` [9]. Linting is a light-weight static analysis that is used to find erroneous code patterns. `ESLint` implements a total of 223 checking rules. We picked the first 10 rules listed on the `ESLint` website and implemented them with `TreeRegexLib`. (We skipped two trivial rules in the list, e.g. `no-debugger`).

For the implementation of the linter, we re-used the converter from our branch/statement-coverage case study. For each checking rule we usually wrote 1-10 `transformers`. In all transformers, we had to implement a custom modifier to report warnings and, in some cases, to perform some extra checks on the list of *captures*. Next we describe the implementation of a couple of checking rules.

The `no-cond-assign` rule checks if there is an assignment in a conditional expression. We wrote a `TreeRegex` expression for each of the syntactic statements and expressions that could have a conditional expression. One such expression is `(%while ((*@ = @*)) @%)`. This `TreeRegex` expression matches a while-loop statement if its condition contains an assignment.

The `no-cond-constant` rule checks if there is a constant expression in a condition. For this checker we modify the `TreeRegex` expressions from the above rule to extract the conditional expression from a condition statement. The conditional expression is then passed through a set of 4 `transformers` which evaluate an expression to 1 (to denote that the expression is a constant) if both of its operands are constant. If the extracted conditional expression evaluates to a constant, we report a warning.

Overall, we managed to implement all the checkers in significantly fewer number of lines than that in `ESLint`. The size of `ESLint` checkers ranges from 33 to 133 lines (median 57 lines), whereas the size of corresponding `TreeRegex` checkers ranges from 1 to 19 lines (median 8 lines) [9]. We also found that the `TreeRegex` expressions used in these checkers are often easy to read and understand. More data is available in Appendix F.

6.4 Finding Errors in C Programs

In this case study, we re-implemented potential-error checkers for C programs [21] using `TreeRegexLib`. The checkers check for the following patterns:

(1) Repeated if branches: `if (x) {y;} else {y;}`
(2) Suspicious loop conditions: `for (i=0; i>0; i++)`
(3) Repeated lines (without side effects): `x; x;`

The first two analyses—identifying repeated branches and suspicious loop conditions—are re-implementations of previous work, and were originally written in a framework designed for creating checkers [21]. This framework also supports five other checkers, four of which reason about the program's control flow graph and are therefore out of scope. The last checker inspects arbitrarily nested if statements; this requires a more complicated expression than is ideal for regex-like tools generally. The two checkers that we *do* reimplement, however, require fewer total lines of code than they do in the original system while having the same functionality.

Though the checks in this section seem simple, they detect errors that may cause serious problems in practice—not crashing bugs, but bugs that silently compute an incorrect result, which are often harder to diagnose [16]. We run all three checks on Linux 4.4 driver code (`gpu`, `net`, and `staging`) and detect 25 bugs and nine false positives. We now describe the implementation of one of these checkers. We describe the remaining checkers in Appendix G.

Repeated if branches. This checker extracts both branches of if-statements and check whether they are syntactically equal. It detects nineteen bugs, thirteen suspicious statements, and six false positives in Linux driver code. The following is one such bug:

```
// linux/drivers/net/wireless/realtek/
// rtlwifi/btcoexist/halbtcoutsrc.c:224
if (priv->mac80211.link_state >= MAC80211_LINKED)
    undec_sm_pwdb = priv->dm.undec_sm_pwdb;
```

[8]In order to convert the SQL commands to serialized trees, we used an available SQL ANTLR grammar [4] and implemented a library that takes the ANTLR parse tree and produces serialized trees.

[9] Note that `ESLint` supports a few options per rule, which turns the checking of the rule on or off. We do not implement such options. If we ignore the lines of code that check options, the number of lines of code for the `ESLint` checkers would still be at least half.

```
else /* associated entry pwdb */
    undec_sm_pwdb = priv->dm.undec_sm_pwdb;
```

The comment above the else branch is a tipoff that this code snippet is a real bug: "associated entry pwdb" describes the false branch but *not* the true branch, which suggests that the two branches should perform different actions.

To implement the checker, we first adapt a parser to output the serialized tree format from the source code, minus any comments or formatting that we don't need. We also write a `TreeRegex` expression that captures if statements: `(%if (@) @ else @%)` [10]. Then, we write a `transformer` with a modifier that compares the matches for the true and false branches of the if statement.

6.5 A Simple Compiler

In this section, we implement a compiler to understand the applicability of `TreeRegexLib` to implement a DSL compiler. The compiler takes `BC--`, a mini-language based on GNU BC [2], and outputs MIPS assembly code. `BC--` supports expressions, functions, loops, and branch statements like those in C, but restricts all variable to have the 32-bit integer type. The complete grammar is given in Appendix H. The front-end (tree-serializer) was implemented on top of the GNU BC [2] parser by adding 195 lines of modifications. The compiler is implemented in 142 lines of C++ code, including 39 `transformers`.

During this case study, we found that `TreeRegexLib` provides many advantages. First, `TreeRegexLib` uses a flexible intermediate representation: mixing the source and target languages in a single serialized tree is allowed, as is adding macros to defer compilation steps. `TreeRegexLib` also helps with debugging. Because of the unified intermediate representation, we could print the intermediate result at any step of compilation without implementing a dedicated pretty printer. Finally, we found that `TreeRegexLib` naturally supports modular development. We can write one transformation step at a time and selectively enable or disable it.

Example transformers. We now describe two example `transformers` to give a sense of how compilers can be implemented with `TreeRegexLib`. A larger selection of the compilation `transformers` are provided in Appendix I.

The example `transformers` handle assignment statements. Assignments have two variations in `BC--`: binary-operation assignments (e.g. `a+=3`) and simple assignments (e.g. `x=5`). During compilation, we first de-sugar binary-operator assignments to simple assignments using using the following `TreeRegex` expression and replacement string: `(%IR (%IR ⦅.+⦆%) ⦅.+⦆= @%)` and `(%IR (%IR $1%)=(%IR(%IR load $1 %) $2 $3%)%)`. We use `IR` to denote serialized trees that are not MIPS, and `MIPS` for the remainder. Here `$1` is the variable name, `$2` is the binary operation and `$3` is the right-hand side of the binary-operation assignment. We use a `load` macro here to defer the compilation of loading the `$1` variable. These macros are later compiled into the appropriate MIPS assembly by another `transformer`. Here we take advantage of the lack of a rigid grammar to mix `BC--` and macros in a single serialized tree.

[10]The parser used in this case study removes comments and normalizes white-space. We additionally have a parser that maintains comments and white-space.

BC-- Size (kB.)	BC-- Parse Time (sec.)	Compile Time (sec.)	MIPS Size (kB.)	Compile Rate (kB./sec.)
1.61	0.001	0.012	14.38	134.17
16.08	0.002	0.082	143.06	196.1
32.16	0.004	0.174	286.62	184.83
64.32	0.008	0.312	573.86	206.15
96.48	0.011	0.467	861.14	206.6
128.64	0.015	0.646	1149.71	199.13
160.8	0.019	0.777	1438.81	206.95
Avg.	-	-	-	190.56

Table 1: BC-like Compiler Performance.

After de-sugaring, assignments are only in one form. We now rewrite simple assignments to a simpler form using another macro, `save`. We use the following `TreeRegex` expression and replacement string: `(%IR (%IR ⦅.+⦆%) = @%)` and `(%MIPS $2 (%IR save $1%)%)`. This replacement string places the expression on the right-hand side of the assignment first (`$2`), and then a `save` macro for the variable (`$1`). The right-hand side expression will be compiled by another `transformer`. The compiled MIPS code maintains the invariant that, when an expression is evaluated, it stores the result in an *accumulator* register. The `save` macro takes this value and stores it in the appropriate place in memory, and will be compiled a later `transformer`. As demonstrated here, we were able to modularize the compilation of assignments in separate steps, and each step is implemented independently.

Performance. To investigate the performance of our `TreeRegexLib` implementation, the runtime of the `BC--` compiler was profiled. All measurements were done on a Intel(R) i7-3630QM CPU with 16GB of RAM and a solid state drive (SSD). We used an example BC program—`extensions.bc`—from the X-Bc project [7], edited to only use the features of `BC--`. The file is 181 lines of non-comment source code. To simulate a large, human-written code, the contents of the file were duplicated. Table 1 summarizes our results. The first column shows the size of the source file excluding comments and empty lines. The second column and the third column show the runtime of the `BC--` to serialized tree converter, and the compiler, respectively. The fourth column shows the size of the resulting MIPS code. The final column shows the performance of the compiler by giving the ratio of the first column to the fourth column.

From this result we make two observations. First, the compiler runs adequately fast—it handles the largest input (composed of 18k lines) in under a second. This shows that `TreeRegexLib` could be practical for sophisticated transformation tasks. Second, the compilation rate is consistent: approximately 190 kB. per seconds. This indicates that `TreeRegexLib` has linear time complexity in this case-study.

7 RELATED WORK

Language extensions. Machete [32], *PLT-Redex* [39], and *XDuce* [36] extend programming languages to enable structural pattern matching. Machete extends Java to support structured term patterns, XML patterns, and bit-level patterns. PLT-Redex extends Racket to support structural pattern matching over S-expressions. XDuce is an extension of ML, which uses regular expressions to describe patterns over XML documents. The above projects are bound to a particular problem and a particular programming language. To the contrary, TreeRegex can be applied to any problem domain in any language as long as a parser is provided.

Natural Language Tree Processing. The natural language processing community has developed many tools to match and manipulate natural language parse trees. Tools like Tgrep [49], Tgrep2 [50], and Tregex [40] are designed to search trees representing the natural-language parse of a sentence. These tools effectively operate on a subset of serialized trees by requiring each sub-tree to be labeled with the expression type—in the trees these tools operate on, each sub-tree can and must contain exactly one string, which is the first part of that sub-tree. TreeRegex is thus more general. The tool Tsurgeon [40] additionally allows for modification of the natural language parse trees. Tsurgeon is designed around a different paradigm: TreeRegex replacement operations are performed once and have a known, bounded running time, but Tsurgeon tree manipulation operations may never complete. This is because a Tsurgeon tree manipulation operation continues until no tree or sub-tree is found that matches a the given pattern.

Island Grammars. Island Grammars [26, 42, 59] have been proposed to specify parts of programs satisfying a specific syntax. They have been used as a light-weight means of implementing syntactic analysis tools [43, 55]. Island Grammars view programs as sequences composed of land chunks (parts to find out) and water chunks (parts to ignore), and uses a SGLR parser to find out as many land chunks as possible. Unlike TreeRegex, Island Grammars are not suitable for program transformation because they do not have a mechanism to capture parts parsed as water chunks. Moreover, Island Grammars ignore the structure of the ignored parts, so they usually have false positives.

Perl-style regular expressions. Perl [63] and PCRE [31] allow for named, mutually-recursive patterns, which are sufficient to implement CFG parsers. Perl and PCRE, like CFG parsers, are intended to solve different problems than TreeRegex: TreeRegex attempts to implement a simple tool for matching tree-structured text. Perl and PCRE are at least as powerful as CFG parsers and have significantly more syntactic varieties than TreeRegex. The generality of Perl and PCRE expressions comes at a high performance cost: these systems are Turing-complete and use backtracking, which has exponential runtime for even simple regular expressions. For instance, we attempted to use a Perl/PCRE expression for the motivating example (detecting calls to eval), but the expression required over 3 hours to match on a 300k JavaScript file. The TreeRegexLib implementation finished in 3 seconds.

Rewriting systems. AST-rewriting frameworks [15, 18, 20, 24, 33, 58, 61] synthesize an AST-manipulating program from a high-level description. They have been successfully applied to a variety of problem domains, including optimizing compilers [48] and domain-specific languages [19, 62]. We propose TreeRegex as a complement to AST-rewriting frameworks, and not as a replacement.

The main difference stems from the design principle: our tool aims to provide a regular expression equivalence of AST manipulation, i.e. a simple library concentrating on matching and replacement operations. We believe that TreeRegex is easy to learn because of the small core. TreeRegexLib retains expressiveness by delegating tasks other than matching, such as free-form tree traversal and conditional matching, to host languages without convoluting the core concepts. Moreover, it can be easily implemented for any programming language, as we did this for C++, Java, and JavaScript. AST-rewriting frameworks, on the other hand, are designed to provide a versatile and powerful standalone solution to develop AST-manipulation programs. However, being a standalone solution, inter-operation between an AST-rewriting framework and another programming language may not be as easy as it is for TreeRegexLib. For instance, frameworks such as Cobra [33] design their matching and rewriting syntax to be a super-set of the language that it manipulates (the C-like languages C, C++ and Java). Therefore, most of the work has to be done within the AST-rewriting framework and this can be a burden to newcomers.

In terms of describing patterns to match, AST-rewriting frameworks and TreeRegex again take different routes. In modern AST-rewriting frameworks a matching pattern can be in a concrete syntax form not involving any details of the parser-specific AST representation [60]. TreeRegex expressions, to the contrary, allow users to mix regular-expressions with serialized trees. If we want to find all variable declarations where the variable names begin with an upper-case letter, for example, concrete syntax is not expressive enough. TreeRegex can express such patterns easily.

Rewriting logic [22, 39, 41, 51] is a logic framework to describe and verify semantics and transformation of programs. TreeRegex focuses more on practical issues, such as compilation and syntactic checking, and rewriting-logic aims for more formal problems, such as formal specifications [46].

Tree automata. Tree automata and *tree transducers* have been widely studied [14, 23, 27, 45] and used to form the theoretical basis of many tree manipulating tools [25, 40, 44, 57]. TreeRegex expressions and transformers can be formalized as a means of expressing tree automata and tree transducers.

REFERENCES

[1] Common Vulnerabilities and Exposures the standard for information security vulnerability names. =http://cve.mitre.org/. Accessed:2016-11-14.
[2] Gnu bc. https://www.gnu.org/software/bc/. Accessed:2016-03-22.
[3] Mips32 architecture. https://imgtec.com/mips/architectures/mips32/. Accessed:2016-03-22.
[4] grammars-v4. https://github.com/antlr/grammars-v4. Accessed:2016-11-14.
[5] WordPress.com. =https://wordpress.com/. Accessed:2016-11-14.
[6] Acorn: A tiny, fast JavaScript parser, written completely in JavaScript. https://github.com/ternjs/acorn. Accessed:2016-03-22.
[7] X-bc. http://x-bc.sourceforge.net/index.html. Accessed:2016-07-04.
[8] Calendar Event Multi View. https://wordpress.org/plugins/cp-multi-view-calendar/. Accessed:2016-11-14.
[9] Eslint: The pluggable linting utility for JavaScript and JSX. http://eslint.org/. Accessed:2016-03-22.
[10] esotope: ECMAScript code generator steroids. https://github.com/inikulin/esotope. Accessed:2016-03-22.
[11] Esprima: ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/. Accessed:2016-03-22.
[12] Exploit Database offensive securityâĂŹs exploit database archive. https://www.exploit-db.com/. Accessed:2016-11-14.
[13] istanbul: A JavaScript code coverate tool written in js. https://gotwarlost.github.io/istanbul. Accessed:2016-03-22.
[14] R. Alur and L. D'Antoni. Streaming tree transducers. In *Automata, Languages, and Programming*, pages 42–53. Springer, 2012.
[15] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Term Rewriting and Applications*, pages 36–47. Springer, 2007.
[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static

analysis to find bugs in the real world. *Commun. ACM*, 53(2), 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL http://doi.acm.org/10.1145/1646353.1646374.

[17] P. Bisht, P. Madhusudan, and V. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):14, 2010.

[18] P. Borovanskỳ, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electronic Notes in Theoretical Computer Science*, 15:55–70, 1998.

[19] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *ACM SIGPLAN Notices*, volume 39, pages 365–383. ACM, 2004.

[20] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008.

[21] F. Brown, A. Noetzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[23] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2007.

[24] J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[25] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. In *ACM SIGPLAN Notices*, volume 49, pages 384–394. ACM, 2014.

[26] P. T. Devanbu. Genoa — a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(2):177–212, 1999.

[27] Z. Fülöp and H. Vogler. *Syntax-directed semantics: Formal models based on tree transducers*. Springer Science & Business Media, 2012.

[28] N. Fulton, C. Omar, and J. Aldrich. Statically typed string sanitation inside a Python. In *Proceedings of the 2014 International Workshop on Privacy & Security in Programming*, pages 3–10. ACM, 2014.

[29] S. Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., New York, NY, USA, 1966.

[30] W. G. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[31] P. Hazel. PCRE: Perl compatible regular expressions. *Online http://www.pcre.org*, 2005.

[32] M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages*, pages 150–166. Springer, 2008.

[33] G. J. Holzmann. Cobra: A light-weight tool for static and dynamic program analysis. *Innov. Syst. Softw. Eng.*, 13(1):35–49, Mar. 2017. ISSN 1614-5046. doi: 10.1007/s11334-016-0282-x. URL https://doi.org/10.1007/s11334-016-0282-x.

[34] J. E. Hopcroft and J. D. Ullman. Formal languages and their relation to automata. 1969.

[35] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[36] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[37] ISO. Information technology—Syntactic metalanguage—Extended BNF. ISO 14977:1996, International Organization for Standardization, Geneva, Switzerland, 1996.

[38] S. C. Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.

[39] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. *ACM SIGPLAN Notices*, 47(1): 285–296, 2012.

[40] R. Levy and G. Andrew. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the fifth international conference on Language Resources and Evaluation*, pages 2231–2234. Citeseer, 2006.

[41] J. Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721–781, 2012.

[42] L. Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE, 2001.

[43] L. Moonen. Lightweight impact analysis using island grammars. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 219–228. IEEE, 2002.

[44] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT)*, 5(4):660–704, 2005.

[45] M. Nivat and A. Podelski. *Tree automata and languages*. Elsevier Science Inc., 1992.

[46] D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356. ACM, 2015.

[47] T. Parr and K. Fisher. LL (*): the foundation of the ANTLR parser generator. In *ACM SIGPLAN Notices*, volume 46, pages 425–436. ACM, 2011.

[48] M. Pierre-Etienne, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Compiler Construction*, pages 61–76. Springer, 2003.

[49] R. Pito. Tgrep manual page. *Available from Linguistic Data Consortium*, 1994.

[50] D. L. Rohde. Tgrep2 user manual, 2004.

[51] G. Roşu and T. F. Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[52] M. Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[53] G. Steele. *Common LISP: the language*. Elsevier, 1990.

[54] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[55] N. Synytskyy, J. R. Cordy, and T. R. Dean. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 266–278. IBM Press, 2003.

[56] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[57] A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming*, pages 81–96. Springer, 2006.

[58] M. G. J. van den Brand, A. van Deursen, J. Heering, H. de hong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+ sdf meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370. Springer, 2001.

[59] A. van Deursen, T. Kuipers, and L. Moonen. Arrangement and method for a documentation generation system. *US Patent. Applied Aug*, 2000.

[60] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, pages 299–315, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44284-7. URL http://dl.acm.org/citation.cfm?id=645435.652697.

[61] E. Visser. Program transformation with stratego/xt. In *Domain-specific program generation*, pages 216–238. Springer, 2004.

[62] E. Visser. *WebDSL: A case study in domain-specific language engineering*. Springer, 2008.

[63] L. Wall et al. The Perl programming language, 1994.

[64] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[65] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, volume 6, pages 179–192, 2006.

[66] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102. ACM, 2006.

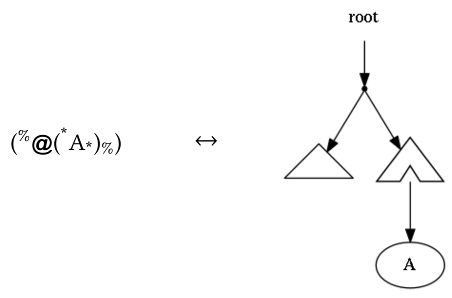


Figure 2: An example TreeRegex expression and its tree representation. Direct expressions are denoted with a dot node, context expressions with a triangle with a smaller triangle removed, subtree expressions with a triangle, and strings as nodes with the string inside.

A TREE FORMULATION OF TREEREGEX EXPRESSION

TreeRegex expressions can also be represented as a tree. Figure 2 shows an example tree for the TreeRegex expression $(^{\%}$ @ $(^{*}$A$*)$ $_{\%})$. We use a triangle to denote a wildcard expression, a notched triangle to denote a context expression, and a dot to represent an exact expression.

B FORMAL DESCRIPTION OF THE TREEREGEX MATCHING ALGORITHM

A formal description of the function *match(t, s)* can be found in Figure 3. The figure describes five rules, corresponding to the four cases from Section 3.4, except that context matches are described in two rules. The rules should be read as follows. The statements above the horizontal bar are hypotheses of the rule; if the hypotheses are satisfied, then the statement below the bar is true. The behavior of *match* is described recursively using the rules. A statement of the form $match(t, s) \longrightarrow K$ states that $match(t, s)$ returns the list K (which is assumed to be not *nil*). $regexMatch(t, w) \longrightarrow K$ means that regular expression t matches the string w and generates the list K of *captures*. The predicate $isRegex(t)$ is true if t is a pure regular expression.

C FORMAL DESCRIPTION OF THE TREEREGEX REPLACEMENT ALGORITHM

Figure 4 summarizes the replacement rules. In these rules, w refers to a string and s refers to a serialized tree. Any serialized tree is a string, but not all strings are serialized trees (as serialized trees follow a grammar). c is a serialized tree context.

D COMPLEXITY ANALYSIS

We describe two different complexities for matching: one that naïvely uses a significant amount of auxiliary memory and one that uses $O(k)$ auxiliary memory, which we have implemented.

To understand the complexity with higher auxiliary memory, consider each subsequent call to *match* from an initial call to *match*.

$$\textsc{Exact}\ \frac{\begin{array}{c} match(t_1, s_1) \to K_1 \\ \vdots \\ match(t_n, s_n) \to K_n \end{array}}{match((^{\%}t_1 \ldots t_{n\%}), (^{\%}s_1 \ldots s_{n\%})) \to K_1 \cdot \ldots \cdot K_n}$$

$$\textsc{Wildcard}\ \frac{}{match(@, (^{\%}s_1 \ldots s_{n\%})) \to [(^{\%}s_1 \ldots s_{n\%})]}$$

$$\textsc{ContextCapture}\ \frac{match((^{\%}t_1 \ldots t_{n\%}), (^{\%}s_1 \ldots s_{m\%})) \to K}{match((^{*}t_1 \ldots t_{n*}), (^{\%}s_1 \ldots s_{m\%})) \to [\bullet] \cdot K}$$

$$\textsc{ContextBuild}\ \frac{\begin{array}{c} \exists_{i \in 1 \ldots m} match((^{*}t_1 \ldots t_{n*}), s_i) \to [e] \cdot K \\ \nexists_{j \in 1 \ldots m} j < i \wedge match((^{*}t_1 \ldots t_{n*}), s_j) \to [e'] \cdot K' \\ c \overset{def}{=} (^{\%}s_1 \ldots s_{i-1} e s_{i+1} \ldots s_{m\%}) \end{array}}{match((^{*}t_1 \ldots t_{n*}), (^{\%}s_1 \ldots s_{m\%})) \to [c] \cdot K}$$

$$\textsc{StringMatch}\ \frac{\begin{array}{c} isRegex(t) \\ regexMatch(t, u) \to K \end{array}}{match(t, u) \to K}$$

Figure 3: Algorithm for matching a TreeRegex expression to a serialized tree. Each t_i is a TreeRegex or regular expression. Each s_i is a serialized tree and each w is a string

$$\textsc{Context}\ \frac{\begin{array}{c} K(n) = c \\ c(s) = s' \end{array}}{replace(w_1\$nsw_2, K) \to w_1 s' w_2}$$

$$\textsc{Tree}\ \frac{K(n) = s}{replace(w_1\$nw_2, K) \to w_1 s w_2}$$

$$\textsc{String}\ \frac{K(n) = w'}{replace(w_1\$nw_2, K) \to w_1 w' w_2}$$

Figure 4: Algorithm for replacements. Each w is a string and each s is a serialized tree. c is a serialized tree context.

Firstly, note each subsequent call to *match* reduces the input either into a sub-TreeRegex expression or sub-serialized tree expression. Also note that all *match* calls on a TreeRegex expression that have no sub-TreeRegex expressions terminate, as do all the calls on strings (i.e. not serialized trees) in serialized trees. This makes sense—each call to *match* makes progress along either the TreeRegex expression or the serialized tree. Let us assume that these cases—matching on a TreeRegex expression with sub-expressions and matching on a string from a serialized tree—take constant time. If they take constant time and we do not repeat computation (calling *match* on the same argument pair twice), then the runtime is bounded by the number of sub-TreeRegex expression and sub-serialized trees. Let us use n and m to refer to these two values; the big-Oh, assuming we do not repeat computation, is $O(mn)$. This is saying that, at worst, each sub-TreeRegex expression can be matched against each sub-serialized tree.

To prevent repeated computation, we can naïvely use memoization—everytime a matching is performed on a sub-TreeRegex expression and sub-serialized tree pair, we record the result. This leads to $O(mn)$ auxiliary memory at minimum. We found that, in practice, a slower algorithm that uses less memory was sufficient and quite usable. The slower algorithm algorithm uses $O(k)$ auxiliary memory. We next analyze the complexity of this slower algorithm.

The algorithm presented in Section 3.4 can repeatedly compute matches: during the matching on a context expression $(^{*}(^{*}t_{*})_{*})$ and a serialized tree s, each subtree of s can be matched on t multiple times in the worst case. The call to *match* with $(^{*}(^{*}t_{*})_{*})$ will result in subsequent calls to *match* with $(^{*}t_{*})$ and calls with t. Both of these subsequent calls will be done on the input argument s and each of its subtrees s_i. Hence, *match(t, s_i)* is called at least twice in this example, because both levels of context expressions backtrack. We bound the computation caused by k context expressions as traversing the entire serialized tree and applying each `TreeRegex` expression on each element of the serialized tree. Because each `TreeRegex` expression may contain a context expression, we may evaluate all `TreeRegex` expressions and serialized tree match calls k times. This leads to a runtime that is bounded by $O((mn)^{k+1})$, while using $O(k)$ auxiliary memory. We need $O(k)$ memory to keep track of the sub-serialized tree index while matching context expressions. This runtime was practical because the number of context expressions is typically 1 or 2, at most, in our usage.

E TRANSFORMATION ALGORITHM AND `TRANSFORMERS` EXAMPLE

E.1 Pseudocode of the Transformation Algorithm

Given a list of user-defined `transformers`, say T, and a serialized tree, say s, the following pseudocode shows the steps of the algorithm:

```
function traverse(s)
    let σ be a user-defined state

    // apply all pre transformers
    foreach (type, t, M, r) in T
        if type = pre
            K ←match(t, s)
            K ← M(K, σ)
            if K ≠ nil and r is defined
                s ←replace(r, K)
    // recursively traverse all children
    foreach s_i where s is of the form (%s_1 ... s_n%)
        if s_i is a serialized tree
            s_i ← traverse(s_i)
    s ← (%s_1 ... s_n%)
    // apply all post transformers
    foreach (type, t, M, r) in T
        if type = post
            K ←match(t, s)
            K ← M(K, σ)
            if K ≠ nil and r is defined
                s ←replace(r, K)
    return s
```

E.2 `Transformer` example

We illustrate the power of `transformers` by example: we implement an interpreter for an expression language. Consider the following simple expression language:

⟨*expr*⟩ ::= ⟨*expr*⟩ + ⟨*expr*⟩ | ⟨*id*⟩ | ⟨*int*⟩ | let ⟨*id*⟩ = ⟨*int*⟩ in ⟨*expr*⟩

⟨*id*⟩ ::= a sequence of letters

⟨*int*⟩ ::= a sequence of digits

Expressions are the addition of two expressions, identifiers, integers, or let-in expressions, defining an integer constant. The “`let`” ambiguity can be handled in any way for the purposes of this example. The following is an example expression in this language: “`let x = 1 in let y = 2 in x + let x = 3 in x + y + 3`”. This expression evaluates to 9, by replacing identifiers with their aliased integers and performing additions.

In order to implement an interpreter for the language, we use a list of four `transformers`. The simplest `transformer` we need is the one that handles integer additions. The `TreeRegex` expression for addition expressions is `(`$^{\%}$ `⦅\d+⦆ \+ ⦅\d+⦆` $_{\%}$`)`. We define modifier *MAdd* to do the addition as follows. (Here we use a JavaScript-like pseudo language to illustrate the modifiers.)

```
function MAdd(K, σ)
  return [K(1) + K(2)]
```

The replacement string for this `transformer` is just the computed value, so it is `$1`. If this `transformer` has the post-order type, then it will collapse any serialized trees that do not use identifiers into their values. For example, consider the post-order application of this `transformer` to “$(^{\%}(^{\%}3{+}4_{\%}){+}(^{\%}5{+}6_{\%})_{\%})$”. First, the transformer will run on the sub-serialized trees “$(^{\%}3{+}4_{\%})$” and “$(^{\%}5{+}6_{\%})$” and rewrite them into “7” and “11”, respectively. The serialized tree after this transformation is “$(^{\%}7{+}11_{\%})$”. After another application to the only serialized tree, the result will be “18”.

Handling identifiers requires two steps; we need to collect identifier values and we need to replace them. In our serialized tree representation, we surround identifiers with “$(^{\%}$” and “$_{\%})$”. To collect the mapping from identifiers to integers, we need to match on “`let`” expressions and use the `TreeRegex` expression `(`$^{\%}$`let (`$^{\%}$`⦅.+⦆`$_{\%}$`) = ⦅\d+⦆ in @`$_{\%}$`)`. We can specify that the state passed to the modifier function keeps track of the mapping as follows:

```
function MLet(K, σ)
  pushMapping(σ, K(1) ↦ K(2))
  return nil
```

The modifier uses the state σ to keep track of the mapping of identifiers to integers, by pushing the identifier mapping for the “`let`” expression. This modifier returns *nil*, so the serialized tree is never modified by it (so a replacement string is not used). This `transformer` provides information for subsequent `transformers`, and we assign it a pre-order type so that it is performed before them.

We use the mapping produced by this modifier in the `transformer` for identifiers. We match on `(`$^{\%}$ `⦅[a-z]+⦆` $_{\%}$`)`, and use the following modifier:

```
function MId(K, σ)
  return [lookup(σ, K(1))]
```

This modifier uses a lookup function to find the last pushed integer mapped to the identifier, and the integer is returned as a captured value. The replacement string is, as before, just `$1`. This transformer does not rely on information from sub-serialized trees, so it can have any order type.

Checker name	TreeRegex LOC	ESLint LOC
no-cond-assign	8	133
no-console	1	56
no-constant-cond	19	73
no-control-regex	2	57
no-dupe-args	10	73
no-dupe-keys	15	48
no-duplicate-case	8	33
no-empty	1	46
no-empty-character-class	1	45
no-extra-boolean-cast	13	78

Table 2: JavaScript Linter: `TreeRegex` vs. `ESLint` LOC

Now that we have transformers that collect identifier mappings and use them, we only need to be concerned with restoring mappings after a "`let`" expression is finished. To do this we use a `transformer` that matches on the "`let`" expressions (which we did not modify in the other "`let`" `transformer`). We can use a similar `TreeRegex` expression to last time: `(%let (%⦅.+⦆%) = ⦅\d+⦆ in ⦅\d+⦆ %)`. We use the following modifier to remove that identifier mapping:

```
function MClearLet(K, σ)
  popMapping(σ)
  return K
```

Finally, we use the replacement string `$3` to rewrite the "`let`" into just the integer after the "`in`". This `transformer` must be applied after the "`in`" expression has been evaluated (a post-order type).

F JAVASCRIPT LINTER DATA

Table 2 shows, for each rule, the number of lines of JavaScript code to implement a checker in comparison to the number of lines used by the corresponding checker in `ESLint`.

Overall, we managed to implement all the checkers in significantly fewer number of lines than that in `ESLint`. The size of `ESLint` checkers ranges from 33 lines to 133 lines (median 57 lines), whereas the size of corresponding `TreeRegex` checkers ranges from 1 line to 19 lines (median 8 lines) [11]. We also found that the `TreeRegex` expressions used in these checkers are often easy to read and understand.

G ADDITIONAL C CHECKERS

Suspicious loop conditions. This checker flags instances where a loop's condition and increment do not match. We detect four odd loop increment-bound pairs: (>, ++), (>=, ++), (<, --), (<=, --). These describe instances where, for example, the loop index is supposed to be less than a bound but is decremented on each loop iteration.

```
linux/drivers/net/ethernet/qlogic
/netxen/netxen_nic_hw.c:2334
// int k, u32 read_cnt;

for (k = 0; k < read_cnt; k--) {
    nx_rd_dump_reg(read_addr,
        adapter->ahw.pci_base0, &read_value);
    *data_buff++ = read_value;
    read_addr += read_stride;
}
```

[11]

in this case, the loop terminates when `k < read_cnt`. unfortunately, int `k` is initialized to 0 and *decremented* on each iteration of the loop; after the first iteration of the loop, `k` will be negative. `read_cnt`, on the other hand, is guaranteed to be positive (since it is unsigned). as a result, `k` will never be less than `read_cnt`. the loop will proceed until `k` underflows; since `int` underflow is undefined behavior, there are no guarantees about what happens to `k` *or* the loop after many hundreds of iterations.

We implement this checker using different `TreeRegex` expressions to extract different suspicious loop bound-increment pairs. The following `TreeRegex` expression, for example, detects (<, --):`(% for(%(@; (%@ < @%); (% @--%)) @%)`. We use a `transformer` to ensure that the variable on the left-hand side of the bound is the same variable that is incorrectly incremented or decremented. This checker detects two true bugs in `gpu`, one in `net`, and one in `staging`. A similar expression is used to detect the same issue for pre-decrement in loops. None of its reports are false positives.

Repeated lines. This checker emits a warning when two lines of code are directly repeated. We filter any lines with calls repeated separately more than twice (e.g. `x; x; y; x; x;`), since we want to identify mis-types and incomplete copy-pastes, not instances where developers are purposely using the same calls multiple times. We also filter `read`, `write`, `in`, and `out` for the same reasons. We implement this checker with the `TreeRegex` expression `(%(%@; %)@;%)` to capture the consecutive expressions, then we use the modifier to check both **@** matches for equality. It filters out cases with obvious side effects like `x++;`. The checker identifies two bugs, two suspicious statements, and three false positives—two of which are commented as "workarounds."

H BC-- LANGUAGE GRAMMAR

Grammar for BC-like language to compiled, shortened for clarity and exposition, is as follows:

⟨*program*⟩ ::= ⟨*function*⟩$^+$

⟨*function*⟩ ::= '`define`' ⟨*id*⟩ '(' [⟨*id*⟩[',' ⟨*id*⟩]*]$^?$ ')' ⟨*block*⟩

⟨*block*⟩ ::= '{' [⟨*statement*⟩ ';']$^+$ '}'

⟨*statement*⟩ ::= '`break`' | ⟨*expr*⟩ |'`if`' '(' ⟨*expr*⟩ ')' ⟨*block*⟩
 | '`while`' '(' ⟨*expr*⟩ ')' ⟨*block*⟩ | '`return`' '(' ⟨*expr*⟩ ')'

⟨*expr*⟩ ::= ⟨*id*⟩ | ⟨*int*⟩ | '(' ⟨*expr*⟩ ')' | ⟨*unary-op*⟩ ⟨*expr*⟩
 | ⟨*id*⟩ ⟨*assign-op*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*bin-op*⟩ ⟨*expr*⟩
 | ⟨*id*⟩ '(' [⟨*expr*⟩ [',' ⟨*expr*⟩]*]$^?$ ')'

⟨*assign-op*⟩ ::= ⟨*bin-op*⟩ '=' | '='

⟨*bin-op*⟩ ::= '+' | '-' | '*' | '/' | '^' | '==' | '!='

⟨*unary-op*⟩ ::= '-' | '!'

⟨*id*⟩ ::= a non-empty string of alphanumeric characters

⟨*int*⟩ ::= a non-empty string of numeric characters

I ADDITIONAL COMPILER `TRANSFORMERS`

⟨*expr*⟩ '+' ⟨*expr*⟩ *Expressions.* This expression sums the results of two other expressions. We implement this as a post-order `transformer`, similar

[11] Note that `ESLint` supports a few options per rule, which turns the checking of the rule on or off. We do not implement such options. If we ignore the lines of code that check options, the number of lines of code for the `ESLint` checkers would still be at least half of the number of LOC reported in the table.

to the example in Section E.2. The following `TreeRegex` expression matches these expressions: `(%IR @ \+ @%)`. This pattern states that we are looking for an expression of two sub-serialized trees, separated by a non-meta-character plus. If these two sub-serialized trees contain the code to generate the value of the left and right expressions, we just need to use these values in the summation. Unfortunately, the evaluations of both expressions return their results in the accumulator register, so we need to save the result after evaluating one operand expression. We can use the `push`, `top`, and `pop` macros to handle the necessary stack operations. If we use *$a0* as the accumulator register and *$t0* as a temporary register to store popped value, our replacement string is: `(%MIPS $1 (%IR push /$a0%) $2 (%IR top /$t0%) add /$a0 /$t0 /$t0 (%IR pop%)%)`. We escape `$` with forward slashes to distinguish it from the `$` meta-character. In this replacement string, we place the code to evaluate the left expression first, then we save the current value of the accumulator register to the stack, followed by the second expression's code, we retrieve the value of the first expression, and, finally, we perform the addition. The last part restores the stack to its original height by popping off the value we pushed.

Collecting Local Variable Offsets. Local variables can only be introduced as parameters to a function in this language and they are stored on the stack. The offsets in the stack can be determined by the location of the variable in the parameter list: the last parameter is 4 bytes from the frame pointer, the second-to-last parameter is 8 bytes from the frame pointer (the size of a 32-bit integer farther than the last parameter), and so on. We implemented a `transformer` computing the relative offset of the variable by visiting parameters in a sequence. This `transformer` must run before other `transformers` on the body of the function. The non-terminal for parameters uses a Kleene star and we encode this in a binary tree form in the serialized tree. For instance, if the parameters of a function were "`a,b,c`", the serialized tree would be: `(%IR (%IR parameter a%),(%IR (%IR parameter b%),(%IR (%IR parameter c%)%)%)%)`. To match this and other parameter serialized trees, we use the following `TreeRegex` expressions: `(%IR (%IR parameter ⟪.+⟫ %),@%)` and `(%IR (%IR parameter ⟪.+⟫ %)%)`.

The modifier function tracks variable offsets using the state:

```
function MParam(K, σ)
  v ← lastMapping(σ)
  pushMapping(σ, K(1) ↦ v+4)
  return nil
```

This modifier function takes the name of the variable and maps it to the last mapping plus the 4 bytes. If there is no last mapping, then the `lastMapping` function returns zero. The modifier function returns *nil*, so the serialized tree is unchanged.

Save Macro. The `save` macro is an example of the modularity of `transformers` and their usefulness during development. Before implementing the `save` macro, we could visually inspect the serialized tree after the existing `transformers` had run to see if there were errors. Once we had finished implementing the `save` macro logic, we just added the `transformer`. The `save` macro is transformed using the `TreeRegex` expression `(%IR save ⟪ .* ⟫ %)`. The `save` `transformer` uses the following modifier function to lookup the variable offset for the saved variable and put it as a captured value:

```
function MVar(K, σ)
  return [lookup(σ, K(1))]
```

The captured value is evaluated with the following replacement string, which generates a save from the accumulator to the offset from the frame pointer register: `(%MIPS sw /$a0 $1 (/$fp)%)`. This `transformer` is very simple because it was able to be separated from the handling of the assignment syntax.

Variable Loading and the Load Macro. The second macro that uses the variable offsets is the `load` macro. This macro is generated from variable expressions using another simple order-agnostic `transformer`. The `TreeRegex` expression is `(%IR (%IR ⟪.+⟫ %)%)`, which finds variable expressions and captures the name of the variable. The replacement string generates a `load` macro of the captured variable (the modifier function does no modification of the captured values): `(%IR load $1%)`.

The `load` macro is handled in a nearly identical order-agnostic `transformer` to the `save` macro. It is made up of the following parts: the `TreeRegex` expression `(%IR load ⟪.+⟫ %)`, the *MVar* modifier function from before, and the replacement string `(%MIPS lw /$a0 $1(/$fp)%)`, which generates the load from the correct offset from the frame-pointer register.

Clearing Local Variable Offsets. The last part of handling local variables is forgetting the offsets when we have finished with the function. To do this, we write a post-order `transformer` on functions. We can match with the following `TreeRegex` expression: `(%IR define .* (@)@%)` and use the following modifier function to forget the mappings:

```
function MFunction(K, σ)
  clearAllMappings(σ)
  return nil
```

The modifier function returns *nil*, so there is no need for a replacement string. We could also erase the function at this point; instead the implementation erases all non-"`MIPS`" serialized trees before printing the final MIPS code.

'if' Statements. A 'if' statement semantically involves the evaluation of a predicate, a test on that predicate, and possibly the execution of a body. To generate code for this, we need to use labels after this body: we need to know where to jump if the predicate evaluates to false. We can write a `transformer` that does this; the modifier function can generate the necessary label. Because MIPS assembly does not allow for duplicate labels, we must use the state to generate unique labels. The `TreeRegex` expression for this `transformer` matches 'if' statements and is: `(%IR if(@)@%)`. In this `TreeRegex` expression, the first captured value is the predicate and the second is the body that may be evaluated if the predicate is true. We use these captured values in the replacement string, so our modifier function must leave them intact and just provide a label. We can use the following modifier function to do that:

```
function MLabel(K, σ)
  v ← nextUniqueLabel(σ)
  append(K, v)
  return K
```

This modifier function appends the next unique label—the resulting captured values can be used with the following replacement string to generate the MIPS code: `(%MIPS $1(%MIPS beqz /$a0 $3%) $2 (%MIPS $3:%)%)`. This replacement string places the code for the predicate first, followed by a test on the predicate's result in the accumulator, and then either branches to the label, or proceeds to the body's code. The label is placed after the body, so that branching to the label will skip over the body. This `transformer` is order-agnostic because it does not generate any sub-serialized trees that are not MIPS code already.