Refactoring Regular Expressions

Carl Chapman
Department of Computer Science
Iowa State University
carl1976@iastate.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
ktstolee@ncsu.edu

ABSTRACT

Regexes are hard to understand. Let me tell you how.

1. INTRODUCTION

Regular expressions are used frequently by developers for many purposes, such as parsing files, validating user input, or querying a database. Regexes are also employed in MySQL injection prevention [?] and network intrusion detection [?]. However, recent research has suggested that regular expressions (regexes) are hard to understand, hard to compose, and error prone |cite. Given the difficulties with working with regular expressions and how often they appear in software projects and processes, it seems fitting that efforts should be made to ease the burden on developers.

Tools have been developed to make regexes easier to understand, and many are available online. Some tools will, for example, highlight parts of regex patterns that match parts of strings as a tool to aid in comprehension.[1] Others will automatically generate strings that are matched by the regular expessions [?]. Other tools will automatically generate regexes when given a list of strings to match [?, ?]. The commonality of such tools provides evidence that people need help with regex composition and understandability.

In software, code smells have been found to hinder understandability of source code []. Once removed through refactoring, the code becomes more understandable, easing the burden on the programmer. In regular expressions, such code smells have not yet been defined, perhaps in part because it is not clear what makes a regex smelly.

As with source code, in regular expressions, there are multiple ways to express the same semantic concept. For example, the regex, `'aa*'` matches an `a` followed by zero or more `a`'s, and is is equivalent to `'a+'` , which matches one or more `a`'s. What is not clear is which representation, `'aa*'` or `'a+'`, is preferred. Preferences in regex refactorings could come from a number of sources, including which is easier to maintain, easier to understand, or better conforms to community standards, depending on the goals of the programmer.

In this work, we introduce possible refactorings in regular expressions by identifying equivalence classes of regex representations and transformations between the representations. These equivalence classes provide options for how to represent double-bounds in repetitions (e.g., `'a{1,2}'` or `'a|aa'`), single-bounds in repetitions (e.g., `'a{2}'` or `'aa'`), lower bounds in repetitions (e.g., `'a{2,}'` or `'aaa*'`), character classes (e.g., `'[0-9]'` or `'[\d]'`), and literals (e.g., `'\a'` or `'\x07'`). We suggest directions for the refactorings, for example, from `'aa*'` to `'a+'`, based on two high-level concepts: which representation appears most frequently in source code (conformance to community standards) and which is more understandable by programmers, based on comprehension tests completed by 180 study participants. Our results identify canonical representations for four of the five equivalence classes based on mutual agreement between community standards and understandability. For the fifth group on double-bounded repetitions, two recommendations are given depending on the goals of the programmer.

Our contributions are:

- Identification of equivalence classes for regular expressions with possible transformations within each class,
- Conducted an empirical study with 180 participants evaluating regex understandability,
- Conducted an empirical study identifying opportunities for regex refactoring in Python projects based on how regexes are expressed, and
- Identified canonical regex representations that are the most understandable and conform best to community standards, backed by empirical evidence.

To our knowledge, this is the first work to apply refactoring to regular expressions. Further, we approach the problem of identifying preferred regex representations by looking at thousands of regexes in Python projects and measuring the understandability of various regex representations using human participants. The rest of the paper describes equivalence classes and possible refactorings (Section 2), research questions (Section 3), the study of regex representations in Python projects (Section 4), and the regex understandability study using human participants (Section 5). We discuss the overall analysis results in Section 6, implications in Section 7, related work in regexes (Section 8), and conclude in Section 9. **TODO.LAST: can remove for space**

[1] https://regex101.com/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

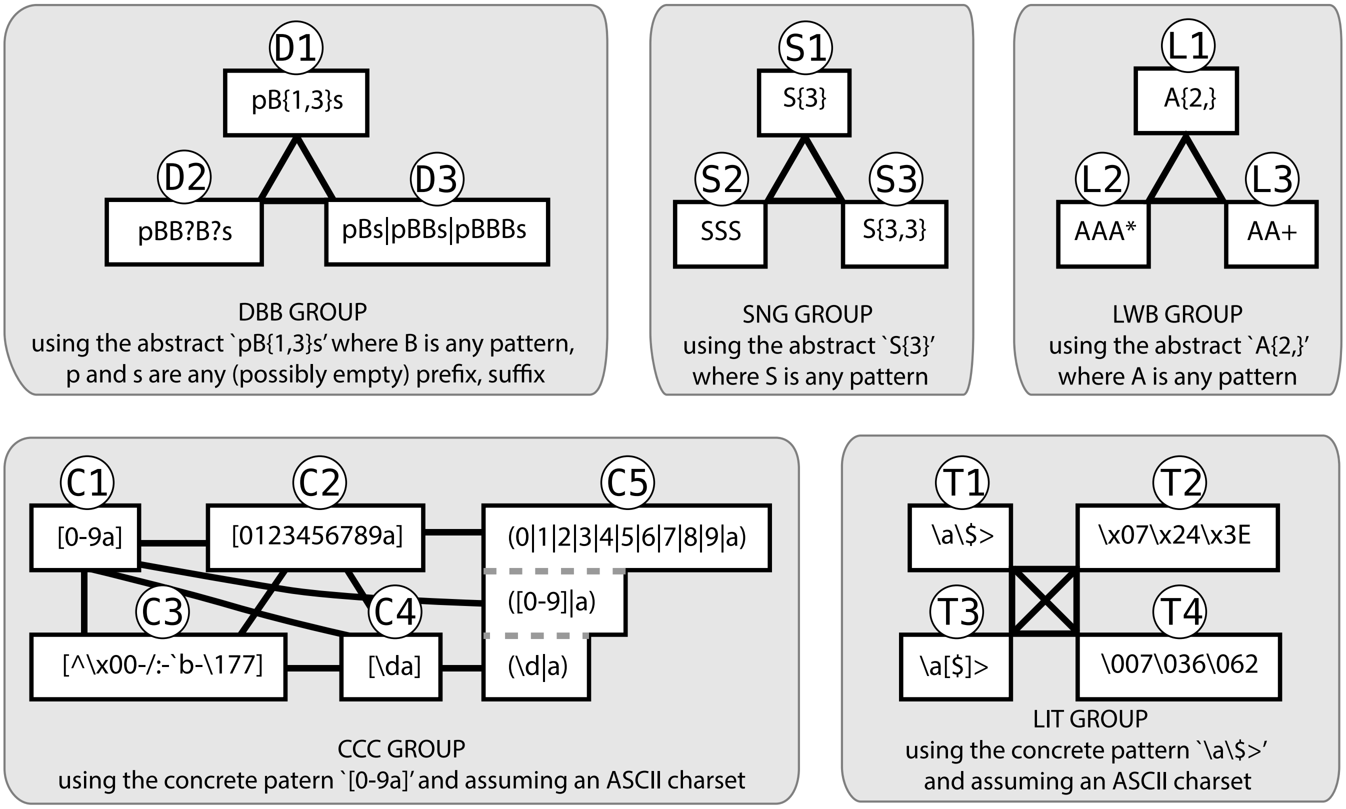

Figure 1: Equivalence classes with various representations of semantically equivalent refactorings within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

2. REFACTORINGS

After studying over 13,000 distinct regex strings from nearly 4,000 Python projects[2], we have defined a set of equivalence classes for regexes with refactorings that can transform among members in the classes. For example, `AAA*` and `AA+` are semantically identical, except one uses the star operator (indicating zero or more repetitions) and the other uses the plus operator (indicating one or more repetitions). Both match strings with two or more `A`'s.

Figure 1 displays the five equivalence classes in grey boxes and various semantically equivalent *representations* of a regex are shown in white boxes. For example, LWB is an equivalence class with representations that all have a lower bound on repetitions. Regexes `AAA*` and `AA+` are both members of this class mapping to representations L2 and L3, respectively, along with the L1 representation, `A{2,}`. The undirected edges between the representations define possible refactorings. Identifying the best direction for each arrow in the possible refactorings is discussed in Section **??**.

We use concrete regexes in the representations to more clearly illustrate examples of the representations. However, the `A`'s in the LWB group abstractly represent any pattern that could be operated on by a repetition modifier (literal characters, character classes, groups, etc.). We chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero. Next, we describe each group, the representations, and possible transformations in detail:

[2]same dataset used in prior work [**?**]

CCC Group.

The Custom Character Class (CCC) group has regex representations that use the custom character class language feature or can be represented by such a feature. A custom character class enables a programmer to specify a set of alternative characters, any of which can match. For example, the regex `'c[ao]t'` will match both the string "cat" and the string "cot" because, between the `c` and `t`, there is a custom character class, `[ao]`, that specifies either `a` or `o` (but not both) must be selected. We use the term *custom* to differentiate these classes created by the user from the default character classes, : `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`, provided by most regex libraries.

Next, we provide descriptions of each representation in this equivalence class:

C1: Any pattern using a range feature like `[a-f]` as shorthand for all of the characters between 'a' and 'f' (inclusive) within a (non-negative) character class belongs to the C1 node.

C2: Any pattern that contains at least one (non-negative) custom character class without any shorthand representations, specifically ranges or defaults. For example, `'[012]'` is in C2, but `'[0-2]'` is not.

C3: Any character classes expressed using negation, which is indicated by a caret (i.e., `^`) followed by a custom character class specification (including literal characters, default character classes and ranges). For example the pattern `[^ao]` matches every character *except* a or o. If the applicable character set is known (ie. ASCII, UTF-8, etc.), then any non-negative character class can be represented as a negative character class. For example, assuming an ASCII charset for simplicity (which has 128 characters: `\x00-\x7f`), a character class representing the lower half: `[\x00-\x3f]` can be represented by negating the upper half: `[^\x40-\x7f]` (notice the caret).

C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class belongs to the C4 node. Note that a pattern can belong to both C1 and C4, such as `[a-f\d]`. The edge between C1 and C4 represents the opportunity to express the same pattern as `[a-f0-9]` by transforming the default digit character class into a range (or visa-versa). This transformed version would only belong to the C1 node.

C5: While not expressed using a character class, these representations can be transformed into custom character classes by removing the ORs and adding square brackets (e.g., `(\d|a)` in C5 is equivalent to `[\da]` in C4). All custom character classes expressed as an OR of length-one sequences, including defaults or other CCCs, are included in C5. Note that because an OR cannot be directly negated, it does not make sense to have an edge between C3 and C5 in figure 1. However, any regex belonging to C3 can transition to C1, C2 or C4 and then transition to C5 from there.

DBB Group.

The Double-Bounded (DBB) group contains all regex patterns that use some repetition defined by a (non-equal) lower and upper boundary. For example the pattern `pB{1,3}s` represents a `p` followed by one to three sequential `B` patterns, then followed by a single `s`. This will match "pBs", "pBBs", and "pBBBs".

D1: Any pattern that uses the curly brace repetition with a lower and upper bound, such as `pB{1,3}s`, belongs to the D1 node. Note that `pB{1,3}s` can become `pBB{0,2}s` by pulling the lower bound out of the curly braces and into the explicit sequence (or visa versa). Nonetheless, it would still be part of D1, though this within-node refactoring on D1 is not discussed in this work.

D2: Any pattern that uses the questionable (i.e., `?`) modifier implies a lower-bound of zero and an upper-bound of one, and belongs to D2. For example, when a double-bounded regex has zero on the lower bound, as is the case with `pBB{0,2}s` in D1, transforming it to D2 involves replacing the curly braces with n questionable modifiers, where n is the upper bound, creating `pBB?B?s`.

D3: Any pattern that has a repetition with a lower and upper boundary and is expressed using ORs is part of D3. The example, `pB{1,3}s` would become `pBs|pBBs|pBBs` by expanding on each option in the boundaries. Note also that a pattern can belong to multiple nodes in the DBB group, for example, `(a|aa)X?Y{2,4}` belongs to all three nodes.

Note that a pattern can belong to multiple nodes in the DBB group, for example, `(a|aa)X?Y{2,4}` belongs to all three nodes: `Y{2,4}` maps it to D1, `X?` maps it to D2, and `(a|aa)` maps it to D3.

LIT Group.

All patterns that are not purely default character classes have to use some literal tokens to specify what characters to match. In Python and most other languages that support regex libraries, the programmer is able to specify literal tokens in a variety of ways. In our example we use the ASCII charset, in which all characters can be expressed using hex and octal codes like `\xF1`, and `\0108`, respectively. This group defines transformations among various representations of literals.

T1: Patterns that do not use any hex characters (T2), wrapped characters (T3) or octal (T4), but use at least one literal character belong to the T1 node.

T2: Any pattern using hex tokens, such as `\x07`, belongs to the T2 node.

T3: Any literal wrapped in square brackets belongs to T3. Literal character can be wrapped in brackets to form a custom character class of size one, such as `[x][y][z]`. This style is used most often to avoid using a backslash for a special character in the regex language, for example, `[{]` which must otherwise be escaped like `\{`.

T4: Any pattern using octal tokens, such as `\007`, belongs to the T4 node.

Patterns often fall in multiple of these representations, for example, `abc\007` includes literals a, b, and c, and also octal `\007`, thus belonging to T1 and T4.

LWB Group.

The lower-bounded (LWB) group contains all patterns that specify only a lower boundary on the number of repetitions required for a match. This can be expressed using curly braces with a comma after the lower bound but no upper bound, for example `A{3,}` which will match 'AAA', 'AAAA', 'AAAAA', and any number of A's greater or equal to 3.

L1: Any pattern using this curly braces-style LWB repetition belongs to node L1.

L2: The kleene star (KLE) means zero-or-more of something, and so `X*` is equivalent to `X{0,}`. Any pattern using KLE belongs to the L2 node.

L3: One of the most commonly used regex features is additional repetition (ADD), for example `T+` which means one-or-more T's. This is equivalent to `T{1,}`. Any pattern using ADD repetition belongs to the L3 node.

Regex patterns often belong to multiple nodes, for example, with `A+B*`, `A+` maps it to L3 and `B*` maps it to L2. We note that the refactorings from L1 to L3 and L2 to L3 are not always possible, specifically when the lower bound is zero and the pattern is not repeated in sequence (e.g., '`A*`' or '`A{0,}`').

SNG Group.

This equivalence class contains three representations of a regex that deal with repetition of a single element in the regex, represents by `S`.

S1: Any pattern with a single repetition boundary in curly braces belongs to S1. For example, `S{3}`, states that S appears exactly three times in sequence.

S2: Any pattern that is explicitly repeated two or more times and could use repetition operators is part of S2.

S3: Any pattern with a double-bound in which the upper and lower bounds are same belong to S3. For example, `S{3,3}` states `S` appears a minimum of 3 and maximum of 3 times.

The important factor distinguishing this group from DBB and LWB is that there is a single finite number of repetitions, rather than a bounded range on the number of repetitions (DBB) or a lower bound on the number of repetitions (LWB).

Example.

Regular expressions will often belong to many representations in the equivalence classes described here, and often multiple representations within an equivalence class. Using an example from a Python project, the regex `'[^ ]*\.[A-Z]{3}'` is a member of S1, L2, C1, C3, and T1. This is because `'[^ ]'` maps it to C3, `'[^ ]*'` maps it to L2, `'[A-Z]'` maps it to C1, `'\.'` maps it to T1, and `'[A-Z]{3}'` maps it to S1. As examples of refactorings, moving from S1 to S2 would be possible by replacing `'[A-Z]{3}'` with `'[A-Z][A-Z][A-Z]'` and moving from L2 to L1 would replace `'[^ ]*'` with `'[^ ]{0,}'`, resulting in a refactored regex of: `'[^ ]{0,}\.[A-Z][A-Z][A-Z]'`.

3. RESEARCH QUESTIONS

After defining the equivalence classes and potential regex refactorings as described in Section 2, we wanted to know which representations in the equivalence classes are considered desirable and which might be smelly. Desirability for regexes can be defined many ways, including maintainable and understandable. As prior work has shown that regexes are difficult to read [], we seek to define refactorings toward understandability.

We define understandability two ways. First, assuming that common programming practices are more understandable than uncommon practices, we explore the frequencies of each representation from Figure 1 using thousands of regexes scraped from Python projects. Second, we then present people with regexes exemplifying some of the more common characteristics and ask them comprehension questions along two directions: determine which of a list of strings are matched by the regex, and compose a string that is matched by the regex.

Our overall research questions are:

RQ1: Which refactorings have the strongest *community support* based on how frequently each representation appears in regexes in open source Python projects?

RQ2: Which refactorings have the strongest support based on *understandability* as measured by matching strings and composing strings?

RQ3: Which regex representations are most desirable based on both community support and understandability?

4. COMMUNITY SUPPORT STUDY (RQ1)

To determine how common each of the regex representations is in the wild, we collected regexes from GitHub projects. We specifically targeted Python projects as it is

```
               function              pattern                  flags
r1 = re.compile('(0|-?[1-9][0-9]*)$', re.MULTILINE)
```

Figure 2: Example of one regex utilization

a popular programming language with a strong presence on GitHub. Further, Python is the fourth most common language on GitHub (after Java, Javascript and Ruby) and Python's regex pattern language is close enough to other regex libraries that our conclusions are likely to generalize.

4.1 Artifacts

A regex utilization is one single invocation of a regex library. Figure 2 presents an example of one regex utilization from Python, the language used in our artifact analysis (Section 4), with key components labeled. The *function* called is `re.compile`. The *pattern* used to define what strings this utilization will match is `(0|-?[1-9][0-9]*)$`. The *flag* `re.MULTILINE` modifies the rules used by the regex engine when matching. When executed, this utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. The pattern in Figure 2 will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Our goal was to collect regexes from a variety of projects to represent the breadth of how developers use the language features. Using the GitHub API, we scraped 3,898 projects containing Python code. We did so by dividing a range of about 8 million repo IDs into 32 sections of equal size and scanning for Python projects from the beginning of those segments until we ran out of memory. At that point, we felt we had enough data to do an analysis without further perfecting our mining techniques. We built the AST of each Python file in each project to find utilizations of the `re` module functions. In most projects, almost all regex utilizations are present in the most recent version of a project, but to be more thorough, we also scanned up to 19 earlier versions. The number 20 was chosen to try and maximize returns on computing resources invested after observing the scanning process in many hours of trial scans. All regex utilizations were obtained, sans duplicates. Within a project, a duplicate utilization was marked when two versions of the same file have the same function, pattern and flags. In the end, we observed and recorded 53,894 non-duplicate regex utilizations in 1,544 projects. **TODO.LAST: 1544 may be a white lie...the 13K+ patterns come from 1544 projects, but the 54k utilizations (before pruning) probably come from something like 1900 projects, and that number is somewhere in the git history of tour de source**

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of utilizations using flags, which can alter regex behavior. An additional 6.5% of utilizations contained patterns that could not be compiled because the pattern was non-static (e.g., used some runtime variable). The remaining 80.8% (43,525) of the utilizations were collapsed into 13,711 distinct pattern strings. Each of the pattern strings was preprocessed by removing Python quotes (`'\\W'` becomes `\\W`), unescaping escaped characters (`\\W` becomes `\W`) and parsing the resulting string using an ANTLR-based, open source

Table 1: How frequently is each alternative expression style used?

Node	Description	Example	nPatterns	% patterns	nProjects	% projects
C1	char class using ranges	`'^[1-9][0-9]*$'`	2,479	18.2%	810	52.5%
C2	char class explicitly listing all chars	`'[aeiouy]'`	1,903	14.0%	715	46.3%
C3	any negated char class	`'[^A-Za-z0-9.]+'`	1,935	14.2%	776	50.3%
C4	char class using defaults	`'[-+\d.]'`	840	6.2%	414	26.8%
C5	an OR of length-one sub-patterns	`'(@|<|>|-|!)'`	245	1.8%	239	15.5%
D1	curly brace repetition like {M,N} with M<N	`'^x{1,4}$'`	346	2.5%	234	15.2%
D2	zero-or-one repetition using question mark	`'^http(s)?://'`	1,871	13.8%	646	41.8%
D3	repetition expressed using an OR	`'^(Q|QQ)\<(.+)\>$'`	10	.1%	27	1.7%
T1	no HEX, OCT or char-class-wrapped literals	`'get_tag'`	12,482	91.8%	1,485	96.2%
T2	has HEX literal like `\xF5`	`'[\x80-\xff]'`	479	3.5%	243	15.7%
T3	has char-class-wrapped literals like [$]	`'[$][{]\d+:([^}]+)[}]'`	307	2.3%	268	17.4%
T4	has OCT literal like `\0177`	`'[\041-\176]+:$'`	14	.1%	37	2.4%
L1	curly brace repetition like {M,}	`'(DN)[0-9]{4,}'`	91	.7%	166	10.8%
L2	zero-or-more repetition using kleene star	`'\s*(#.*)?$'`	6,017	44.3%	1,097	71.0%
L3	one-or-more repetition using plus	`'[A-Z][a-z]+'`	6,003	44.1%	1,207	78.2%
S1	curly brace repetition like {M}	`'^[a-f0-9]{40}$'`	581	4.3%	340	22.0%
S2	explicit sequential repetition	`'ff:ff:ff:ff:ff:ff'`	3,378	24.8%	861	55.8%
S3	curly brace repetition like {M,M}	`'U[\dA-F]{5,5}'`	27	.2%	32	2.1%

PCRE parser[3]. This parser was unable to support 0.5% (73) of the patterns due to unsupported unicode characters. Another 0.2% (25) of the patterns used regex features that we chose to exclude because they appeared very rarely (e.g., reference conditions). An additional 0.1% (16) of the patterns were excluded because they were empty or otherwise malformed so as to cause a parsing error. After removing all problematic patterns as described, 13,597 distinct patterns from 1,544 projects remained to be used in this study.

4.2 Metrics

We measure community support by matching each regex in the corpus to the representations (nodes) in Figure 1 and counting the number of *patterns* that contain the representation and the number of *projects* that contain the representation. A *pattern* is extracted from a utilization, as shown in Figure 2. Note that a regex can belong to multiple representations, and a regex can belong to multiple projects since we collapsed duplicates and only analyze the 13,711 distinct regex patterns that represent 43,525 regex utilizations across the projects.**TODO.MID: feels weird to hear this again right away, maybe simplify the metrics paragraph**

4.3 Analysis

To determine how many of the representations match patterns in the corpus, we performed an analysis using the PCRE parser and by representing the regexes as token streams, depending on the characteristics of the representation. Our analysis code is available on GitHub[4]. Next, we describe the process in detail:

4.3.1 Presence of a Feature

For the representations that only require a particular feature to be present, such as the question-mark in D2, the features identified by the PCRE parser were used to decide membership of patterns in nodes. These feature-requiring nodes are as follows: D1 requires double-bounded repetition with different bounds, D2 requires the question-mark repetition, S1 requires single-bounded repetition, S3 requires double-bounded repetition with the same bounds, L1 requires a lower-bound repetition, L2 requires the kleene star (`*`) repetition, L3 requires the add (`+`) repetition, and C3 requires a negated custom character class.

4.3.2 Features and Pattern

For some representations, the presence of a feature is not enough to determine membership. However, the presence of a feature and properties of the pattern can determine membership.

Identifying D3 requires an OR containing at least two entries - some sequence present in one entry repeated N times, and then the same sequence present in another entry repeated N+1 times. This is a hard pattern to detect directly, but we identified candidates by looking for a sequence of N repeating groups with an OR-bar (ie. `|`) next to them on one side (either side). This produced a list of 113 candidates which we narrowed down manually to 10 actual members.

Identifying T2 requires a literal feature that matches the regex `(\\x[a-f0-9A-F]{2})` which reliably identifies hex codes within a pattern. Similarly T4 requires a literal feature and must match the regex `((\\0\d*)|(\\\d{3}))` which is specific to Python-style octal, requiring either exactly three digits after a slash, or a zero and some other digits after a slash. Only one false positive was identified which was actually the lower end of a hex range using the literal `\0`.

Identifying T3 requires that a single literal character is wrapped in a custom character class (a member of T3 is always a member of C2). T1 requires that no characters are

[3]https://github.com/bkiers/pcre-parser
[4]https://github.com/softwarekitty/regex_readability_study

wrapped in brackets or are hex or octal characters, which actually matches over 91% of the total patterns analyzed.

4.3.3 Token Stream

The following representations were identified by representing the regex patterns as a sequence of dot-delimited tokens. Identifying S2 requires any element to be repeated at least twice. This element could be a character class, a literal, or a collection of things encapsulated in parentheses. Identifying C1 requires that a non-negative character class contains a range. Identifying C2 requires that there exists a custom character class that does not use ranges or defaults. Identifying C4 requires the presence of a default character class within a custom character class, specifically, `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`. Identifying C5 requires an OR of length-one sequences (literal characters or any character class).

4.4 Results

Table 1 presents the frequencies with which each representation appears in a regex pattern and in a project scraped from GitHub. The *node* column references the representations in Figure 1 and the *description* column briefly describes the representation, followed by an *example* from the corpus. The *nPatterns* column counts the patterns that belong to the representation, followed by the percent of patterns out of 13,597. The *nProjects* column counts the projects that contain a regex belonging to the representation, followed by the percentage of projects out of 1,544. Recall that the patterns are all unique and could appear in multiple projects, hence the project support is used to show how pervasive the representation in across the whole community. For example, 2,479 of the patterns belong to the C1 representation, representing 18.2% of the patterns. These appear in 810 projects, representing 52.5%. Representation D1 appears in 346 (2.5%) of the patterns but only 234 (15.2%) of the projects. In contrast, representation T3 appears in 39 *fewer* patterns but 34 *more* projects, indicating that D1 is more concentrated in a few projects and T3 is more widespread across projects.

Using the pattern frequency as a guide, we can create refactoring recommendations based on community frequency. For example, since C1 is more prevalent than C2, we could say that C2 is smelly since it could better conform to the community standard if expressed as C1. Thus, we might recommend a $\overrightarrow{C2C1}$ refactoring. Based on patterns alone, the winning representations per equivalence class are C1, D2, T1, L2, and S2. With one exception, these are the same for recommendations based on projects. The difference is that L3 appears in more projects than L2, so it is not clear which would be more desirable based on community standards. Section 6 explores these results more deeply.

5. UNDERSTANDABILITY STUDY (RQ2)

The overall idea of this study is to present programmers with one of several representations of semantically equivalent regexes and ask comprehension questions. By comparing the understandability of semantically equivalent regexes that have different representations, we aim to understand which representations are more desirable and which are more smelly. This study was implemented on Amazon's Mechanical Turk with 180 participants. Each regex pattern was evaluated by 30 participants. The patterns used were designed to belong to various representations in Figure 1.

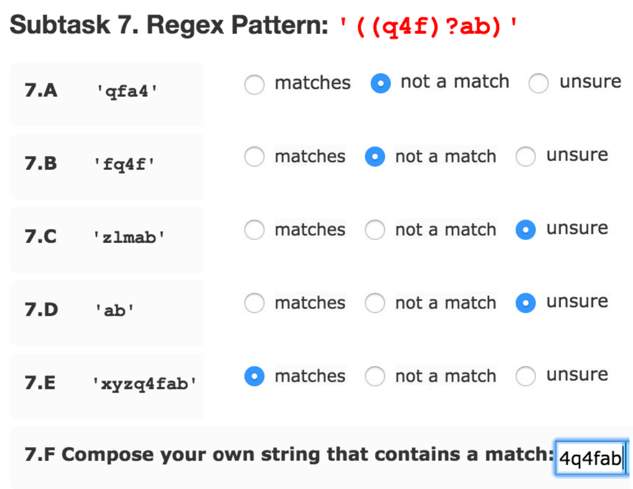

Figure 3: Example of one HIT Question

Table 2: Matching metric example

String	'RR*'	Oracle	P1	P2	P3	P4
1	"ARROW"	✓	✓	✓	✓	✓
2	"qRs"	✓	✓	✗	✗	?
3	"R0R"	✓	✓	✓	?	-
4	"qrs"	✗	✓	✗	✓	-
5	"98"	✗	✗	✗	✗	-
	Score	1.00	0.80	0.80	0.50	1.00

✓ = match, ✗ = not a match, ? = unsure, – = left blank

5.1 Metrics

We measure the understandability of regexes using two complementary metrics, *matching* and *compostition.*

Matching: Given a pattern and a set of strings, a participant determines which strings will be matched by the pattern. There are four possible responses for each string, *matches*, *not a match*, *unsure*, or blank. An example from our study is shown in Figure 3.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example, consider regex pattern `'RR*'` and five strings shown in Table 2, and the responses from four participants in the *P1*, *P2*, *P3* and *P4* columns. The oracle has the first three strings matching since they each contain at least one `R` character. *P1* answers correctly for the first three strings but incorrectly thinks the fourth string matches, so the matching score is $4/5 = 0.80$. *P2* incorrectly thinks that the second string is not a match, so they also score $4/5 = 0.80$. *P3* marks 'unsure' for the third string and so the total number of attempted matching questions is 4 instead of 5. *P3* is incorrect about the second and fourth string, so they score $2/4 = 0.50$. For *P4*, we only have data for the first and second strings, since the other three are blank. *P4* marks 'unsure' for the second matching question so only one matching question has been attempted, and it was answered correctly so the matching score is $1/1 = 1.00$.

Blanks were incorporated into the metric because questions were occasionally left blank in the study. Unsure responses were provided as an option so not to bias the results when participants were honestly unsure of the answer. These situations did not occur very frequently. Only 1.1% of the responses were left blank and only 3.8% of the responses were marked as unsure.

Composition: Given a pattern, a participant composes a string they think it matches. If the participant is accurate and the string indeed is matched by the pattern, then a composition score of 1 is assigned, otherwise 0. For example, given the pattern `'(q4fab|ab)'` from our study, the string, "xyzq4fab" matches and would get a score of 1, and the string, "acb" does not match and would get a score of 0.

To determine a match, each pattern was compiled using the *java.util.regex* library. A *java.util.regex.Matcher* `m` object was created for each composed string using the compiled pattern. If `m.find()` returned true, then that composed string was given a score of 1, otherwise it was given a score of 0.

5.2 Design

This study was implemented on the Amazon's Mechanical Turk (MTurk), a crowdsourcing platform in which requesters can create human intelligence tasks (HITs) for completion by workers. Each HIT is designed to be completed in a fixed amount of time and workers are compensated with money if their work is satisfactory. Requesters can screen workers by requiring each to complete a qualification test prior to completing any HITs.

5.2.1 Worker Qualification

Workers were pre-qualified by answering questions regarding some basics of regex knowledge. These questions were multiple-choice and asked the worker to describe what the following patterns mean: `'a+'`, `'(r|z)'`, `'\d'`, `'q*'`, and `'[p-s]'`. To pass the qualification, workers had to answer four of the five questions correctly.

5.2.2 Tasks

Using the patterns in the corpus as a guide, we created 60 regex patterns that were grouped into 26 semantic equivalence groups and 10 metagroups containing six patterns each. Of the 26 semantic groups, 18 had two equivalent patterns each and eight groups had three equivalent patterns each. Four of the metagroups were composed of two groups of three, and six of the metagroups were composed of three groups of two. For example, one of the groups of size two had patterns: `'([0-9]+)\.([0-9]+)'` belonging to representation C1 and `'(\d+)\.(\d+)'` belonging to representation C4. One of the groups of size three contained `'((q4f){0,1}ab)'` belonging to D1, `'((q4f)?ab)'` belonging to D2, `'(q4fab|ab)'` belonging to D3. For each of the 26 groups of patterns, we created five strings, where at least one matched and at least one did not match. These strings were used to compute the matching metric.

Once all the patterns and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a pattern from each of the 10 metagroups. Randomize the order of these 10 patterns, as well as the order of the matching strings for each pattern. After adding a question asking the participant to compose a string that each pattern matches, this creates one task on MTurk. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs. An example of a single regex pattern, the five matching strings and the space for composing a string is shown in Figure 3.

5.2.3 Implementation

Workers were paid \$3.00 for successfully completing a HIT, and were only allowed to complete one HIT. The average completion time for accepted HITs was 682 seconds (11 mins, 22 secs). A total of 55 HITs were rejected, and of those, 48 were rushed through by one person leaving many answers blank, 4 other HITs were also rejected because a worker had submitted more than one HIT, one was rejected for not answering composition sections, and one was rejected because it was missing data for 3 questions. Rejected HITs were returned to MTurk to be completed by others.

5.3 Participants

In total, there were 180 participants in the study. A majority were male (83%) with an average age of 31. Most had at least an Associates degree (72%) and most were at least somewhat familiar with regexes prior to the study (87%). On average, participants compose 67 regexes per year with a range of 0 to 1000. Fittingly, participants read more regexes than they write with an average of 116 and a range from 0 to 2000.

5.4 Analysis

For each of the 180 HITs, we computed a matching and composition score for each of the 10 regexes, using the metrics described in Section 5.1. This allowed us to compute 30 values for each metric and for each of the 60 regexes. Next, we computed average scores for matching and composition per regex.**TODO.LAST: Mentioning NAs here?**

Each regex was a member of one of 26 groupings of equivalent regexes. These groupings allow pairwise comparisons of the metrics values to determine which representation of the regex was most understandable. Among all the groups, we performed 42 pairwise comparisons of the matching and composition scores (i.e., one comparison for each group of size two and three comparisons within each group of size three). For example, one group of size two had regexes, `RR*` and `R+`, which are equivalent and represent a transformation between L2 and L3. The former had an average matching of 86% and the latter had an average matching of 92%. The average composition score for the former was 97% and 100% for the latter. Thus, the community found `R+` from representation L3 more understandable. There were two other pairwise comparisons performed between the L2 and L3 group, using regexes pair `zaa*` and `za+'`, and regexes pair `\..*` and `\.+'`. Considering all three of these regex pairs, the overall matching average for the regexes belonging to L2 was 0.86 and 0.91 for L3. The overall composition score for L2 was 0.91 and 0.98 for L3. Thus, the community found L3 to be more understandable, from the perspective of both understandability metrics, than L2. This information is presented in summary in Table 3, with this specific example appearing in the E3 row. The *Index* column enumerates all the pairwise comparisons evaluated in this experiment, *Representations* lists the two representations, *Pairs* shows how many comparisons were performed, *Match1* gives the overall matching score for the first representation listed, *Match2* gives the overall matching score for the second representation listed, and $H_0 : \mu_{match1} = \mu_{match2}$ uses the Mann-Whitney test of means to compare the matching scores, and presents the p-

Table 3: Averaged Info About Edges (sorted by lowest of either p-value)

Index	Representations	Pairs	Match1	Match2	$H_0 : \mu_{match1} = \mu_{match2}$	Compose1	Compose2	$H_0 : \mu_{comp1} = \mu_{comp2}$
E1	T1 – T4	2	0.80	0.60	0.001	0.87	0.37	**<0.001**
E2	D2 – D3	2	0.78	0.87	**0.011**	0.88	0.97	0.085
E3	L2 – L3	3	0.86	0.91	**0.032**	0.91	0.98	0.052
E4	C2 – C5	4	0.85	0.86	0.602	0.88	0.95	0.063
E5	C2 – C4	1	0.83	0.92	0.075	0.60	0.67	0.601
E6	D1 – D2	2	0.84	0.78	0.120	0.93	0.88	0.347
E7	C1 – C2	2	0.94	0.90	0.121	0.93	0.90	0.514
E8	T2 – T4	2	0.84	0.81	0.498	0.65	0.52	0.141
E9	C1 – C5	2	0.94	0.90	0.287	0.93	0.93	1.000
E10	T1 – T3	3	0.88	0.86	0.320	0.72	0.76	0.613
E11	D1 – D3	2	0.84	0.87	0.349	0.93	0.97	0.408
E12	C1 – C4	6	0.87	0.84	0.352	0.86	0.83	0.465
E13	C3 – C4	2	0.61	0.67	0.593	0.75	0.82	0.379
E14	S1 – S2	3	0.85	0.86	0.776	0.88	0.90	0.638

values. The last three columns list the average composition scores for the representations and the p-value, also using the Mann-Whitney test of means.

Although we had 42 pairwise comparisons, we had to drop six comparisons due to a design flaw since the patterns performed transformations from multiple equivalence classes. For example, pattern `([\072\073])` is in C2 and T4, and was grouped with pattern `(:|;)` in C5, T1, so it was not clear if any differences in understandability were due to the transformation between C2 and C5, or T4 and T1. However, the third member of the group, `([:;])`, could be compared with both, since it is a member of T1 and C2, so comparing it to `([\072\073])` evaluates the transformation between T1 and T4, and comparing to `(:|;)` evaluates the transformation between C2 and C5. The end result is 36 pairwise comparisons across 14 edges from Figure 1.

5.5 Results

Table 3 presents the results of the understandability analysis. A horizontal line separates the first three edges from the bottom 11. In E1 through E3, there is a statistically significant difference between the representations for at least one of the metrics considering $\alpha = 0.05$. These represent the strongest evidence for suggesting the directions of refactoring based on the understandability metrics we defined. Specifically, $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$ are likely to improve understandability.

We note here that participants were able to select *unsure* when they were not sure if a string would be matched by a pattern (Figure 3). From a comprehension perspective, this indicates some level of confusion and we can use that to further corroborate the understandability analysis.

For each pattern, we counted the number of responses containing at least one unsure. We then grouped these by node and found the average number of unsures per pattern by node (out of 30). A higher number of unsures per node may indicate difficulty in comprehending a pattern from that node. These nodes and their average number of unsure responses are organized by quartile in Table **??**. These results corroborate the refactorings suggested by match analysis for the LIT group ($\overrightarrow{T4T1}$) and the LWB group ($\overrightarrow{L2L3}$), because the more understandable node has the least unsures of its

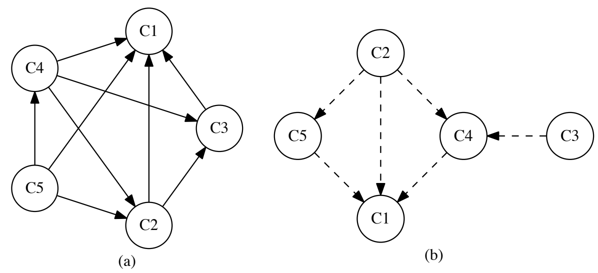

Figure 4: Trend graphs for the CCC equivalence graph: (a) represent the artifact analysis, (b) represent the understandability analysis.

group. The findings for D3 and D2 are contradictory, however and the number of unsures may be too small to indicate anything, except for T2 and T4. The one pattern from T4 that had the most unsures of any pattern (10 out of 30) was ‘`xyz[\0133-\0140]`’, so this may have been the least understandable pattern that we tested.

6. DESIRABLE REPRESENTATIONS (RQ3)

To determine the overall trends in the data, we created total orderings on the representation nodes in each equivalence class (Figure 1) with respect to the community standards (RQ1) and understandability (RQ2) metrics.

6.1 Analysis

At a high level, these total orderings were achieved by building directed graphs with the representations as nodes and edge directions determined by the metrics: patterns and projects for community standards and matching and composition for understandability. Then, within each graph, we performed a topological sort to obtain total node orderings.

The graphs for community support are based on Table 1 and the graphs for understandability are based on Table 3. The following sections describe the processes for building and and topologically sorting the graphs.

6.1.1 Building the Graphs

Table 4: **Average Unsure Responses Per Pattern By Node (fewer unsures on the left)**

	>=Q0(0.67)				>=Q1(1.25)				>=Q2(1.94)				>=Q3(2.54)			
Node	L3	D3	C2	C1	L2	S2	S1	C4	D1	C5	C3	D2	T1	T3	T2	T4
Unsure Responses Per Pattern	0.7	1	1	1	1.3	1.7	1.7	1.9	2	2	2	2.5	2.7	2.7	5.5	8.5

In the community standards graph, we represent a directed edge $\overrightarrow{C2C1}$ when nPatterns(C1) > nPatterns(C2) *and* nProjects(C1) > nProjects(C2). When there is a conflict between nPatterns and nProjects, as is the case between L2 and L3 where L2 is found in more patterns and L3 is found in more projects, an undirected edge $\overline{L2L3}$ is used. This represents that there was no winner based on the two metrics. After considering all pairs of nodes in each equivalence class that also have an edge in Figure 1, we have created a graph, for example Figure 4a, that represents the frequency trends among the community artifacts.

In the understandability graph, we represent a directed edge $\overrightarrow{C2C1}$ when match(C1) > match(C2) *and* compose(C1) > compose(C2). When there is a conflict between match and compose, as is the case with T1 and T3 where match(T1) is higher but compose(T3) is higher, an undirected edge $\overline{T1T3}$ is used. When one metric has a tie, as is the case with composition in E9, we resort to the matching metric to determine $\overrightarrow{C5C1}$. An example understandability graph for the CCC is shown in Figure 4b.

6.1.2 Topological Sorting

Once the graphs are built for each equivalence class and each set of metrics, community standards and understandability, we apply a modified version of Kahn's topological sorting algorithm to obtain a total ordering on the nodes, as shown in Algorithm 1. The first modification is to remove all undirected edges since Kahn's operates over a directed graph.

In Kahn's algorithm, all nodes without incoming edges are added to a set S (Line 5), which represents the order in which nodes are explored in the graph. For each n node in S (Line 6), all edges from n are removed and n is added to the topologically sorted list L (Line 8). If there exists a node m that has no incoming edges, it is added to S. In the end, L is a topologically sorted list.

One downside to Kahn's algorithm is that the total ordering is not unique. Thus, we mark ties in order to identify when a tiebreaker is needed to enforce a total ordering on the nodes. For example, on the understandability graph in Figure 4b, there is a tie between C3 and C2 since both have no incoming edges, so they are marked as a tie on Line 5. Further, when $n = C2$ on line 7, both C5 and C4 are added to S on Line 12, thus the tie between them is marked on line 15. In these cases, a tiebreaker is needed.

Breaking ties on the community standards graph involves choosing the representation that appears in a larger number of projects, as it is more widespread across the community.

Breaking ties in the understandability graph uses the metrics. Based on Table 3, we compute the average matching score for all instances of each representation, and do the same for the composition score. For example, C4 appears in E5, E12 and E13 with an overall average matching score of 0.81 and composition score of 24.3. C5 appears in E4 and E9 with an average matching of 0.87 and composition of 28.28. Thus, C5 is favored to C4 and appears higher in the sorting.

Algorithm 1 Modified Topological Sort

1: $L \leftarrow []$
2: $S \leftarrow []$
3: Remove all undirected edges (creates a DAG)
4: Add all disconnected nodes to L and remove from graph. If there is more than one, mark the tie.
5: Add all nodes with no incoming edges to S. If there is more than one, mark the tie.
6: **while** S is non-empty **do**
7:     remove a node n from S
8:     add n to L
9:     **for** node m such that e is an edge $\overrightarrow{nm}$ **do**
10:         remove e
11:         **if** m has no incoming edges **then**
12:             add m to S
13:         **end if**
14:     **end for**
15:     If multiple nodes were added to S in this iteration, mark the tie
16:     remove n from graph
17: **end while**
18: For all ties in L, use a tiebreaker.

6.2 Results

After running the topological sort in Algorithm 1 with tiebreakers, we have a total ordering on nodes for each graph, shown in Table 5. For example, given the graphs in Figure 4a and Figure 4b, the topological sorts are `C1 C3 C2 C4 C5` and `C1 C5 C4 C2 C3`, respectively.

There is a clear winner in each equivalence class, with the exception of DBB. That is, the node sorted highest in the topological sorts for both the community standards and understandability analyses are C1 for CCC, L3 for LBW, S2 for SNG, and T1 for LIT. After the top rank, it is not clear who the second place winner is in any of the classes, however, having a consistent and clear winner is evidence of a preference with respect to community standards and understandability, and thus provides guidance for potential refactorings.

This positive result, that the most popular representation in the corpus is also the most understandable, makes sense as people may be more likely to understand things that are familiar or well documented. However, while L3 is the winner for the LBW group, we note that L2 appears in slightly more patterns.

DBB is different as the orderings are completely reversed depending on the analysis, so the community standards favor D2 and understandability favors D3. Further study is needed on this, as well as on LBW and SNG since not all nodes were considered in the understandability analysis.

Table 5: Topological Sorting, with the left-most position being highest

	CCC	DBB	LBW	SNG	LIT
Community Standards	C1 C3 C2 C4 C5	D2 D1 D3	L3 L2 L1	S2 S1 S3	T1 T3 T2 T4
Understandability	C1 C5 C4 C2 C3	D3 D1 D2	L3 L2	S2 S1	T1 T2 T4 T3

7. DISCUSSION

7.1 Interpreting Results

TODO.NOW: this one next

7.2 Opportunities For Future Work

Equivalence Class Models.

We looked at 5 equivalence classes, each with 3 to 5 nodes. Future work could study this topic using richer models with more and/or better classes and nodes. For example, in our treatment, we have looked at all ranges as equivalent, all defaults as equivalent, and relied on many such generalizations. But the range `[a-f]` is likely to be more understandable for most people than a range like ``[:-`]``. By creating a more granular model of equivalence classes, and making sure to carefully evaluate alternative representations of the most frequently used specific patterns (like `\\s*` and `.+`), many more strong and useful refactorings could be identified.

In addition to breaking our 5 groups into more specific nodes, future work could model refactorings outside of these groups. We have not determined a list of all possible refactoring groups given the functional variety and significant number of features to consider, but we are aware of a few additional equivalence classes outside of our 5 groups, such as:

Single line option `'''(.|\n)+'''` ≡ `(?s)'''(.)+'''`

Multi line option `(?m)G\n` ≡ `(?m)G$`

Multi line option `(?i)[a-z]` ≡ `[A-Za-z]`

Backreferences `(X)q\1` ≡ `(?P<name>X)q\g<name>`

Word Boundaries `\bZ` ≡ `((?<=\w)(?=\W)|(?<=\W)(?=\w))Z`

We focused on refactorings within a group, treating groups as orthogonal to one another. It would be interesting to see if there is some cooperation between pairs of edges in separate groups by applying more than one type of refactoring at once.

Understandability.

We identified and utilized three new ways to measure understandability of regexes: deciding if certain strings match or not, composing strings that are supposed to match, and measuring the frequency of a regex type in a community. There are many more ways to approach understandability, such as deciding what content is captured by a regex, identifying all the matched substrings in a block of text, deciding which regexes in a set are equivalent, finding the minimum modification to some text so that a given regex will match it, and many more. One of the most straightforward ways to address understandability is to directly ask software professionals which from a list of equivalent regexes they prefer and why. It may also be meaningful to provide some code that exists around a regex as context. The example regexes we used were inspired by real regexes, but at least one side of the refactoring was contrived and we did not focus on any specific community (the 1544 projects we obtained regexes from were randomly obtained). If understandability measurements used regexes sampled from the codebase of a specific community(most frequently observed regexes, most buggy regexes, regexes on the hottest execution paths, etc.), and measured the understanding of programming professionals working in that community, then the measurements and the refactorings they imply would be more likely to have a direct and certain positive impact.

In another study, we did a survey where software professionals indicated that understandability of regexes they find in source code is a major pain point. In this study, our participants indicated that they read about twice as many regexes as they compose. What is the impact on maintainers, developers and contributors to open-source projects of not being able to understand a regex that they find in the code they are working with? Presumably this is a frustrating experience - how much does a confusing regex slow down a software professional? What bugs or other negative factors can be attributed to or associated with regexes that are difficult to understand? How often does this happen and in what settings? Future work could tailor an in-depth exploration of the overall costs of confusing regexes and the potential benefits of refactoring or other treatments for confusing regexes.

Regex Refactoring Applications.

Other opportunities exist to improve the understandability of regexes in existing code bases by looking for some of the less understandable regex representations, which can be thought of as antipatterns, and refactoring to the more common or understandable representations. Building migration libraries is another direction of future work to ease the manual burden of this process, similar in spirit to prior work on class library migration [**?**].

Maintainers of code that is intentionally obfuscated for security purposes may want to develop regexes that they understand and then automatically transform them into the least understandable regex possible.

One fundamental concept that many users of regex struggle to learn is when to use regexes for simple parsing, and when to write a full-fledged parser (for example, when parsing HTML). Regexes that are trying to parse HTML, XML or similar languages could be refactored not into a better regex, but into some code with an equivalent intention that does parsing much better.

Many organizations enforce coding standards in their repositories to ease understandability. Presently, we are not aware of coding standards for regular expressions, but this work suggests that enforcing standard representations for various regex constructs could ease comprehension. Besides comprehension, an organization may want to enforce standards for efficiency, or for compatibility with a regex analysis tool like Z3, HAMPI, BRICS or REX.

7.3 Threats to Validity

7.3.1 Internal

We measure understandability of regexes using two metrics, matching and composition. However, these measures may not reflect actual understanding of the regex behavior. For this reason, we chose to use two metrics and present the analysis in the context of reading and writing regexes, but the threat remains.

TODO.MID: what about the threat of too few examples per node? Didn't cover every edge. Regex set is randomly collected online, not focused on any specific target audience.

We treated unsure responses as omissions and did not count those against the participants. Thus, if a participant answered two strings correctly with match/not match, and marked the other three strings as unsure, then this was 2/2 correct, not 2/5.

7.3.2 External

Participants in our survey came from MTurk, which may not be representative of people who read and write regexes on a regular basis.

The regexes we used in the evaluation were inspired by those commonly found in Python code, which is just one language that has library support for regexes. Thus, we may have missed opportunities for other refactorings based on how programmers use regexes in other programming languages.

Our community analysis only focuses on the Python language. Note that because the vast majority of regex features are shared across most general programming languages (e.g., Java, C, C#, or Ruby), a Python pattern will (almost always) behave the same when used in other languages, whereas a utilization is not universal in the same way (i.e., it may not compile in other languages, even with small modifications to function and flag names). As an example, the `re.MULTILINE` flag, or similar, is present in Python, Java, and C#, but the Python `re.DOTALL` flag is not present in C# though it has an equivalent flag in Java.

8. RELATED WORK

Regular expression understandability has not been studied directly, though prior work has suggested that regexes are hard to read and understand since there are tens of thousands of bug reports related to regular expressions [?]. To aid in regex creation and understanding, tools have been developed to support more robust creation [?] or to allow visual debugging [?]. Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [?, ?].

Regular expression refactoring has also not been studied directly, though refactoring literature abounds [?, ?, ?]. The closest to regex refactoring comes from research toward expediting the processing of regular expressions on large bodies of text [?], which could be thought of as refactoring for performance.

Code smells in object-oriented languages were introduced by Fowler [?]. Researchers have studied the impact of code smells on program comprehension [?, ?], finding that the more smells in the code, the harder the comprehension. This is similar to our work, except we aim to identify which regex representations can be considered smelly. Code smells have been extended to other language paradigms including end-user programming languages [?, ?, ?, ?]. The code smells identified in this work are representations that are not common or not well understood by developers. This concept of using community standards to define smells has been used in other refactoring literature for end-user programmers [?, ?].

Exploring language feature usage by mining source code has been studied extensively for Smalltalk [?], JavaScript [?], and Java [?, ?, ?, ?], and more specifically, Java generics [?] and Java reflection [?]. Our prior work ([?], under review) was the first to mine and evaluate regular expression usages from existing software repositories. The intention of the prior work [?] was to explore regex language features usage and surveyed developers about regex usage. In this work, we define potential refactorings and use the mined corpus to find support for the presence of various regex representations in the wild. Beyond that, we measure regex understandability and suggest canonical representations for regexes to enhance conformance to community standards and understandability.

9. CONCLUSION

Acknowledgements

This work is supported in part by NSF SHF-EAGER-1446932.