An empirical study of regular expression use in practice, sampling from Python projects on Github, leading to new concepts for refactoring regular expressions for readability.

by

Carl Allen Chapman

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Kathryn Stolee, Major Professor
Samik Basu
Tien Nguyen

Iowa State University
Ames, Iowa
2016

DEDICATION

I would like to dedicate this thesis to my mother, who believed in me and supported me through many years on a long winding road leading to a satisfying career. I'd also like to thank my wife Chien Wen Hung and our cat Siva for practical and moral support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Kathryn Stolee for her guidance, patience and support throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Samik Basu and Dr. Tien Nguyen.

ABSTRACT

Abstract

Though regular expressions provide a powerful search technique that is baked into every major language, is incorporated into a myriad of essential tools, and has been a fundamental aspect of Computer Science since Thompson in 1968, no one has ever formally studied how they are used in practice, or how to address challenges in composition and comprehension. This thesis presents the original work of studying a sample of regexes taken from Python projects pulled from Github, determining what features are used most often, defining some categories that illuminate common use cases, and identifying areas of significance for tool builders. Furthermore, this thesis defines an equivalence class model used to explore comprehension of regexes, identifying the most common and most understandable representations of semantically identical regexes, suggesting several refactorings and preferred representations. Opportunities for future work include the novel and rich field of regex refactoring, semantic search of regexes, and further fundamental research into regex usage and understandability.

CHAPTER 1. OVERVIEW

1.1 About this thesis

This thesis will begin by introducing fundamental concepts about regular expressions and nomenclature used. This is followed by a detailed description of the Python Regular Expression features that will be examined in this thesis and a summary of historical milestones. To convey the scope of impact that research into regular expressions has, a small survey of applications and research will be provided. Armed with this overview, the main questions that this thesis will explore will be introduced, followed by a section on related exploratory work. The next N sections will detail the studies conducted to explore the research questions of this thesis. Each of these sections will consist of a description of how the study was designed, followed by a presentation of results and a discussion of implications and opportunities for future work. Each section describing an experiment may depend on the results of previous sections. A final discussion will highlight the most important implications and opportunities for future work already presented, as well as any additional implications or future work not mentioned elsewhere. After a conclusion summarizing everything that has been presented, an appendix of artifacts and a bibliography will complete the thesis.

This thesis will explore many details involving characters, strings and regexes. To reduce confusion due to typesetting issues, and to avoid repeatedly qualifying quoted text with phrases like 'the string' or 'the regex', characters will be surrounded in single quotes like `'c'`, strings will be surrounded by double quotes like `"example string"`, and regexes will be presented within a grey box without any quotes like `a+b*(c|d)e\1f`. All strings in this thesis should be considered 'raw' strings - where in Python and Java source code a literal backslash in a string variable must be escaped so that two backslashes are necessary, only one will be shown

in the text of this thesis. This means that a string presented in this thesis like `"a\dc"` would actually be `"a\\dc"` in source code. Invisible characters such as newline will be represented within strings in gray, like `"first line.\nsecond line."`. Organize Sections and references

1.2 Regular expression basics

1.2.1 Language nomenclature

Regular expression languages are systems for specifying sets of strings. There are many regular expression language *variants* with substantially different behavior, and so the term *regular expression* can only refer to the topic in general. An appropriate prefix must always be added in order to refer to a particular variant (eg. Python Regular Expressions can describe a context free language, but Kleene Regular Expressions cannot). Note that it is grammatically correct to capitalize these proper nouns because they refer to languages.

Each variant uses several features to specify sets of strings in a compact manner. A *pattern* is a string that is parsed according to the feature syntax of a variant into units of meaning called *tokens*. A sequence of feature tokens that is valid in a particular variant will always define a set of strings. This sequence of feature tokens will be called a *regex* (regexes will be the plural form).

Regexes are commonly used to extend keyword search - instead of searching some text for a single keyword, the user can search that text for any string in the set specified by the regex. An *engine* implements the rules of a variant in order to perform searching, replacing and other functions within a computing environment. A single variant may have zero or more engines written for it. An engine *compiles* a pattern into a regex. The behavior of an engine may be modified by flags or options, as described in Section N.

1.2.2 Matching strings defined

In this thesis it is often necessary to describe the outcome of searching a particular string using a particular regex. The terminology used is that a regex *matches* a string if that string contains some substring that is equal to a string specified by the regex. For example, the regex

`abc` matches the entire string `"abc"` but also matches part of `"XabcY"`, and so the regex matches both strings. This regex does not match `"ab"` because no `'c'` is present. When considering if a regex matches a string, it is assumed that no flags are modifying the behavior of the engine unless specified in the regex itself.

This choice of terminology results in the most natural flow of words when discussing the behavior of regexes, but conflicts with the terminology used by several engines. For example, Java's `java.util.regex.Matcher.matches()` function requires the entire string to match in order to return true. Also, Python's `re.match()` function requires the beginning of the string to match in order to regurn a `MatchObject`. Instead, our definition of *match* is closer to Java's `java.util.regex.Matcher.find()` function and Python's `re.search()` function. The definition of *match* used in this thesis is useful because, in general, it is a necessary condition (but not always a sufficient condition) for a regex to *match* a string in order for some function provided by an engine to take action based on the match.

1.2.3 Patterns are not regexes

A particular pattern can specify different regexes in different variants. For example, the pattern `"a\{2\}"` specifies the regex `a{2}` in BRE Regular Expressions (which matches the string `"aa"`), but in Python Regular Expressions the same pattern compiles to the regex `a\{2\}` which matches the string `"a{2}"`. It is also possible for a pattern to be valid and compile to a regex in one variant, but be invalid in another. For example the pattern `"^X(?R)?O$"` compiles to a valid regex in Perl 5.10 that uses recursion to require one or more `'X'` characters followed by exactly the same number of `'O'` characters, so that the string `"XXOO"` will match, but `"XXO"` will not match. Trying to compile this pattern in Python will cause an error.

1.3 Description of studied features

This thesis will focus on the features and syntax described in Table 1.1. A detailed introduction to the functionality of these features as studied in this thesis is provided in this section. The features of Python Regular Expressions that we analyze fall into four categories:

Table 1.1: Reference codes, descriptions and examples of select Python Regular Expression features

code	description	example	code	description	example
Elements			Operators		
VWSP	matches U+000B	`\v`	ADD	one-or-more repetition	`z+`
CCC	custom character class	`[aeiou]`	KLE	zero-or-more repetition	`.*`
NCCC	negated CCC	`[^qwxf]`	QST	zero-or-one repetition	`z?`
RNG	chars within a range	`[a-z]`	SNG	exactly n repetition	`z{8}`
ANY	any non-newline char	`.`	DBB	$n \leq x \leq m$ repetition	`z{3,8}`
DEC	any of: 0123456789	`\d`	LWB	at least n repetition	`z{15,}`
NDEC	any non-decimal	`\D`	LZY	as few reps as possible	`z+?`
WRD	[a-zA-Z0-9_]	`\w`	OR	logical or	`a|b`
NWRD	non-word chars	`\W`	Positions		
WSP	\t \n \r \v \f or space	`\s`	STR	start-of-line	`^`
NWSP	any non-whitespace	`\S`	END	end-of-line	`$`
CG	a capture group	`(caught)`	ENDZ	absolute end of string	`\Z`
BKR	match the i^{th} CG	`\1`	WNW	word/non-word boundary	`\b`
PNG	named capture group	`(?P<name>x)`	NWNW	negated WNW	`\B`
BKRN	references PNG	`(P?=name)`	LKA	matching sequence follows	`a(?=bc)`
NCG	group without capturing	`a(?:b)c`	LKB	matching sequence precedes	`(?<=a)bc`
Options			NLKA	sequence doesn't follow	`a(?!yz)`
OPT	options wrapper	`(?i)CasE`	NLKB	sequence doesn't precede	`(?<!x)yz`

1. Elements are individual characters, character classes and logical groups. Elements can be operated on by operators.
2. Options fundamentally modify the behavior of the engine.
3. Repetition modifiers, implicit concatenation and logical OR are operators. The order of operations is described in Section 1.3.3.3.
4. Positions refer to a position between characters. They make assertions about the string on one or both sides of their position.

1.3.1 Elements

1.3.1.1 Elements: Ordinary characters

Ordinary characters in regexes specify a literal match of those characters, for example `z` matches `"z"` and `"abz"`. Regexes can be concatenated together to create a new regex, so that

`z` and `q` can become `zq`, which matches `"XYzq"` but not `"z"`. Python Regular Expressions[1] use the special characters `.,^,$,*,+,?,{,},[,],(,),|` and `\` to implement the features that allow compact specification of sets of strings. These special characters can be escaped using the backslash to be treated as ordinary characters, for example `zq\$` matches `"zq$"`.

1.3.1.2 Elements: Escaped characters and VWSP

Several characters need be written in Python strings using the backslash. These characters are the backslash:`\\`, bell:`\a`, backspace`\b`, form feed:`\f`, newline:`\n`, carriage return:`\r`, horizontal tab:`\t` and vertical whitespace:`\v`. The vertical tab is rarely used and was examined on its own as an individual feature with the code VWSP.

1.3.1.3 Elements: Character Classes

CCC: A *custom character class* uses the special characters `[` and `]` to enclose a set of characters, any of which can match. For example `c[ao]t` matches the both `"cat"` and `"cot"`. The terminology used in this thesis highlights the difference between a *custom* character class and a *default* character class. Default character classes are built-in to the language and cannot be changed, whereas custom character classes provide the user of regex with the ability to create their own character classes, customized to fit whatever is needed. Order does not matter in a CCC, so `[ab]` is equivalent to `[ba]`.

NCCC: A *negated custom character class* uses the special character `^` as the first character within the brackets of a CCC in order to negate the specified set. For example the regex `c[^ao]t` would *not* match `"cat"` or `"cot"`, but would match `"cbt"`, `"c2t"`, `"c$t"` or any string containing a character other than `` `a' `` or `` `o' `` between `` `c' `` and `` `t' ``. Notice that the exact set of characters specified by a NCCC depends on what charset is being used. NCCCs in Python's `re` module use the Unicode charset. In this thesis the 128 characters of traditional ASCII are used for a charset when explaining a concept, because it makes for more compact examples. For instance the NCCC `[^ao]` excludes 2 characters from a set of 128 characters and will therefore match the remaining 126 characters.

[1] https://github.com/python/cpython/blob/master/Lib/re.py

The caret character can be escaped within a CCC, so that `[\^]` represents the set containing only `^`. If a caret appears after some other character, it no longer needs to be escaped, so the CCC `[x^]` represents the set containing `x` and `^`.

RNG: A *range* provides shorthand within a CCC for the set of all characters in the charset between two characters (including those two characters). So `[w-z]` is equivalent to `[wxyz]`. This feature also works with punctuation or invisible characters, as long as the start of the range occurs before the end of the range. For example the CCC with range `[:-@]` is equivalent to the CCC with no range `[:;<=>?@]`. Note that order of ranges and other characters do not matter, so that `[w-z:-@]` is equivalent to `[:-@w-z]`. The dash character can be included in a CCC, for example `[a-z-]` specifies the lowercase letters and the dash. The NCCC `[^-]` represents all characters except the dash.

ANY: The *any* default character class uses the special character `.` to specify any character except the newline character. For example `a.b` specifies all strings beginning with an `a` and ending with a `b` with exactly one non-newline character between the `a` and `b`, such as `"a2b"`, `"aXb"` or `"a b"`. In Python, the meaning of this character class can be altered by passing the 'DOTALL' flag or using the 's' option so that ANY will also match newlines. When this flag or option is in effect, ANY will match every character in the charset.

DEC: The *decimal* default character class uses the special sequence `\d` to specify digits, and so `\d` is equivalent to `[0-9]`.

NDEC: The *negated decimal* default character class indicated by the special sequence `\D` is simply the negation of the DEC default character class, so `\D` is equivalent to `[^0-9]` or `[^\d]`.

WRD: The *word* default character class uses the special sequence `\w` to specify digits, lowercase letters, uppercase letters and the underscore character. Therefore `\w` is equivalent to `[0-9a-zA-Z_]`.

NWRD: The *negated word* default character class indicated by the special sequence `\W` is simply the negation of the WRD default character class. Therefore `\W` is equivalent to `[^0-9a-zA-Z_]` or `[^\w]`.

WSP: The *whitespace* default character class uses the special sequence `\s` to specify whitespace. Many characters may be considered whitespace, but the definition for this thesis will be the space, tab, newline, carriage return, form feed and vertical tab. This set is based on the POSIX `[:space:]` default character class. Therefore the regexes `\s` and `[ \t\n\r\f\v]` are considered equivalent.

NWSP: The *negated whitespace* default character class indicated by the special sequence `\S` is simply the negation of the WSP default character class. Therefore `\S` is equivalent to `[^ \t\n\r\f\v]]` or `[^\s]`.

1.3.1.4 Elements: Logical groups

CG: The *capture group* feature uses the special characters `(` and `)` to logically group some regex. Other operations treat the contents of the logical group as a single unit, so that all operations within the group are performed before operations outside it are considered. This follows the typical use of parenthesis in algebra as expected. For example consider `A(12|98)`, where the regex `12|98` is treated as one element because it is in a CG. Therefore `A(12|98)` matches `"A12"` or `"A98"`. Without the logical grouping provided by CG, the regex `A12|98` will match `"A12"` or `"98"` - the concatenation of `A` is no longer applied to `98` because it is no longer logically next to the `A`.

In addition to providing logical grouping, the text matched by the contents of the capture group is stored, or 'captured' and can be referred to later in the regex by a back-reference or extracted by a program for any purpose. The captured content is frequently referred to by the number of the capture group like 'group 1' or 'group 2'. For example when `(x*)(y*)z` matches `"AxxyyzB"`, group 1 contains `"xx"`, and group 2 contains `"yy"`. Group 0 contains the entire matched portion of input: `"xxyyz"`.

BKR: The *back-reference* feature uses the special character \ followed by a number 'n' to refer to the captured contents of the nth capture group, as defined by the order of opening parenthesis. For example in `(a.b)\1`, the `\1` is referring to whatever was captured by `a.b`. This regex will match the strings `"aXbaXb"` and `"a2ba2b"` but not `"aXba2b"` because the character matched by ANY in `a.b` is `` `X' `` not `` `2' ``.

PNG: A *Python-style named capture group* uses the syntax `(?P<name>X)` to name a capture group. This is known as Python-style because there are other styles of named capture group such as Microsoft's .NET style and other variants. Python's implementation is noteworthy because it was the first attempt at naming groups. Names used must be alphanumeric and start with a letter.

BKRN: The *back-reference; named* feature uses the special syntax `(?P=N)`, and is a back-reference for content captured by PNG with name 'N'. For example, the regex using PNG and BKRN: `(P<OldGreg>a.b)(?P=OldGreg)` is equivalent to the regex using CG and BKR: `(a.b)\1`.

NCG: The *non-capture group* uses the special syntax `(?:E)` to create a NCG containing element 'E'. A NCG can be used in place of a CG to perform logical grouping without affecting capturing logic. So `(?:a+)(b+)c\1` will match `"abcb"` because the NCG `(?:a+)` is ignored by the BKR, so that the first CG is `(b+)`, which is what is back-referenced by `\1`. In contrast, `(a+)(b+)c\1` would not match `"abcb"` but would match `"abca"` because its first CG is `"(a+)"`

1.3.2 Options

OPT: The *options* feature allows the user to modify the engine's matching behavior within the regex itself, instead of using flag arguments passed to the regex engine. For example the regex `(?i)[a-z]` uses the option `(?i)` which switches on the ignore-case flag, so that this regex will match `"lower"` and `"UPPER"`. Other options include `(?a)`, `(?s)`, `(?m)`, `(?l)` `(?x)` and `(?u)`. In Python Regular Expressions, options can appear anywhere within the regex and will have the same effect.

1.3.3 Operators

1.3.3.1 Operators: Repetition modifiers

ADD: *Additional* repetition uses the special character `+` to specify one or more of an element. For example the regex `z+` describes the set of strings containing one or more `` `z' `` characters, such as `"z"`, `"zz"` and `"zzzzz"`. The regex `.+` applies additional repetition to the ANY character class, matching one or more non-newline characters such as `"7a"` or `"tulip"`. In `(a.b)+`, additional repetition is applied to the CG `(a.b)`. By applying additional repetition to this logical group, `(a.b)+` specifies strings with one or more sequential strings matching the regex in that group, such as `"a2b"`, `"a2baXba*b"` or `"a1ba2ba3ba4b"`.

KLE: *Kleene star* repetition uses the special character `*` to specify zero-or-more repetition of an element. For example the regex `pt*` describes the set of strings that begin with a `` `p' `` followed by zero or more `` `t' `` characters, such as `"p"`, `"ptt"` and `"pttttt"`.

QST: *Questionable* repetition specifies zero-or-one repetition of an element. For example `zz(top)?` matches strings `"zz"` and `"zztop"`.

SNG: *Single-bounded repetition* uses the special characters `{` and `}` containing an integer 'n' to specify repetition of some element exactly n times. For example `(ab*){3}` will match exactly three sequential occurrences of the regex `ab*`, such as `"aaa"` or `"abababb"` but not `"aa"` or `"ababb"`.

DBB: *Double-bounded repetition* uses the special characters `{` and `}` containing integers 'm' and 'n' separated by a comma to specify repetition of some element at least m times and at most n times. For example `(A.X){1,3}` will match one, two or three sequential occurrences of the regex `A.X`, such as `"A7X"`, `"AaXAnX"` or `"A*XAgXAqX"`.

LWB: *Lower-bounded repetition* uses the special characters `{` and `}` containing an integer 'n' followed by a comma to indicate at least n repetitions of an element. For example `(Qt){2,}` will match two or more sequential occurrences of `Qt`, such as `"QtQt"` or `"QtQtQtQt"` but will not match `"Qt"`.

LZY: The *lazy* repetition modifier uses the special character `?` *following another repetition operator* to specify lazy repetition. An example of this syntax is `(a+?)a*b`, where QST is applied to the ADD in `a+` to yeild `a+?`, making ADD lazy instead of greedy. This regex will match `"aab"`, capturing `"a"` in group 1. The regex without LZY is `(a+)a*b` which will also match `"aab"` but will capture `"aa"` in group 1.

1.3.3.2 Operators: Logical OR

OR: An *or* is a disjunction of alternatives, where any of the alternatives is equally acceptable. This feature is specified by the `|` special character. Each alternative can be any regex. A simple example is the regex `cat|dog` which specifies the two strings `"cat"` and `"dog"`, so either of these will match.

1.3.3.3 Order of operations

The order of operations is:

1. repetition features
2. implicit concatenation of elements
3. logical OR

For example, consider the regex `A|BC+`. The ADD repetition modifier takes highest importance, so that this regex is equivalent to `A|B(C+)`. Then implicit concatenation joins the two regex `B` and `(C+)` into `B(C+)`, so the regex is equivalent to `A|(B(C+))`. The last operator to be considered is the logical OR, so that this regex is also equivalent to `(A|(B(C+)))`.

1.3.4 Positions

1.3.4.1 Positions: Anchors

STR: The *start anchor* uses the special character `^` to indicate the position before the first character of a string, so `^B.*` will match every string that starts with `'B'` such as `"Bison"` and `"Bouncy castle"`. If the 'MULTILINE' flag or 'm' option is passed to the regex engine, then STR will match the position immediately after every newline. In this

case this regex will match in two separate places for the string `"Big\nBicycle"` - before the `` `B' `` in `"Big"` and before the `` `B' `` in `"Bicycle"`.

END: The *end anchor* uses the special character `$` to indicate either the position between the last newline and the character before it, or between the end of the string and the character before it if the string does not end in a newline. For example `R$` will match `"abcR"` and `"xyz\nR\n"` but not `"R\nxyz\n"`. The 'MULTILINE' flag or 'm' option also affects the END anchor so that if activated, the string `"R\nxyz\n"` *will* match because there exists a line where `` `R' `` is at the end of a line.

ENDZ: The *absolute end anchor* uses the special sequence `\Z` to indicate the absolute end of the. For example `R\Z` will match `"abcR"` and `"xyz\nR"` but not `"xyzR\n"` or `"Rs"`.

1.3.4.2 Positions: Boundaries

WNW: The *word-nonword* anchor uses the special sequence `\b` to indicate the position between a character belonging to the WRD default character class and belonging to NWRD (or no character, such as the beginning or ending of the string). It doesn't matter if WRD or NWRD comes first, but WNW will only match if the first character is followed by its opposite. This is useful when trying to isolate words, for example the regex `\btaco\b` will match `"taco"` or `"My taco!"` because there is never a word character next to the target word. But the same regex will not match `"catacomb"`, `"\_taco"` or `"tacos"`. The escaped backspace character `` `\b' `` can only be expressed inside a CCC like `[\b]` because this sequence is treated as WNW by default.

NWNW: The *negated word-nonword* anchor uses the special sequence `\B` to indicate the position between either two WRD characters or two NWRD characters (or no character, such as the beginning or ending of the string). The regex `\Btaco\B` matches `"catacomb"` because a word character is found on both sides of wherever the `\B` is. The strings `"tacos"` and `"\_taco"` do not match, though, because in both cases some part of `"taco"` is next to the end of the string.

1.3.4.3 Positions: Lookarounds

LKA: The *lookahead* feature uses the special syntax `(?=R)` to check if regex R matches immediately after the current position. The string matched by R not captured, and is also excluded from group 0. That is why this feature is sometimes called a *zero-width lookahead*. For example `ab(?=c)` matches `"abc"` and has `"ab"` in group 0. Note that a regex like `ab(?=c)d` is valid but does not make sense, because the lookahead and `d` can never both match.

LKB: The *lookback* uses the special syntax `(?<=R)` to check if regex R matches immediately before the current position. As with LKA, the content matched by the LKB is excluded from group 0.

NLKA: A *negative lookahead* uses the special syntax (?R)! to require that regex R *does not match* immediately after the current position. Matched content is excluded from group 0.

NLKB: The *negative lookback* uses the special syntax (?<R)! to require that regex R does not match immediately before the current position. Matched content is excluded from group 0.

1.4 Milestones in regular expression history

1.4.1 Kleene's theory of regular events

In 1943 a model for how nets of nerves might 'reason' to react to patterns of stimulus was proposed in a paper by McCulloch-Pitts. In 1951, Kleene further developed this model with the idea of 'regular events'. In his terminology, 'events' are all inputs on a set of neurons in discrete time, a 'definite event' E is some explicit sequence of events, and a 'regular event' is defined using three operators: 1. logical or, 2. concatenation and 3. the Kleene star which represents zero or more of some definite event. Kleene showed that 'all and only regular events can be represented by nerve nets or finite automata', and went on to show that operations on regular events are closed, and to define an algebra for simplifying regular events. The formulas used to describe regular events were named 'regular expressions' in Kleene's 1956 refined paper.

1.4.2 First regex compiler

Many additional formalisms were built on Kleene's set of three operators, and then around 1967[2], Ken Thompson implemented the first regular expression compiler in IBM 7090 assembly for a version of 'qed' (quick editor) at Bell labs. Existing editors were only able to search and replace using whole words. Thompson's editor was able to search and replace using the features STR, END, ANY, CCC, NCCC and KLE[3] in a new language later known as Simple Regular Expressions (SRE). Although these features provided a useful shorthand, they did not expand the expressiveness of SRE beyond the expressiveness of Kleene Regular Expressions.

1.4.3 Early regular expressions in Unix

Thompson went on to create Unix in 1969 with Dennis M. Ritchie, the core of which was an assembler, a shell and 'ed' - an editor with regular expression search/replace capabilities based on qed. Unix tools grep (1973), sed (1974) and awk (1977) also leveraged regular expression concepts. The feature set of regular expressions evolved over time, and although it is outside the scope of this thesis to capture all details of this evolutionary process, a major milestone was the creation of egrep by Alfred Aho in 1975 which effectively defined Extended Regular Expressions (ERE). This new language added the features CG, BKR, SNG, DBB, LWB, QST, ADD and OR as well as 12 default character classes similar to DEC, WRD, WSP, NDEC, NWRD and NWSP but using syntax like `[:digit:]` instead of the modern `\d` for DEC. The BKR feature is noteworthy in that it is the first feature to extend the set of languages expressible by regular expressions beyond the regular languages described by Kleene Regular Expressions. Aho also wrote fgrep, which is optimized for efficiency instead of expressiveness using the AhoCorasick algorithmREF?.

1.4.4 Maturity of standards

In 1979, Hopcroft and Ullman published the 'Cindarella' textbook covering automata and theory supporting the ERE language (excluding back-references). Perl 2 was released in 1988

[2] https://www.bell-labs.com/usr/dmr/www/qed.html
[3] https://www.bell-labs.com/usr/dmr/www/qedman.html

with some regular expression support, and included shorthand for default character classes like `\d` for DEC. The Perl community significantly boosted the popularity and user base of regular expressions. In 1992, Henry Spencer released `regcomp`, a major regular expression library for C. In the same year, the POSIX.2 standard was also released, officially documenting both Basic Regular Expressions (BRE) and ERE. In 1997, the O'Reilly book 'Mastering Regular Expressions' was first published, and the PCRE standard was first released. By the time Perl 5.10 was released in 2007, many advanced features had been introduced like recursion, conditionals and subroutines.

CHAPTER 2. RELATED WORK

2.1 Applications of regex

2.1.0.1 Everyday searching and replacing

rewrite this to cover ephemeral stuff like find/replace in text editors, IDEs, Browsers, etc. Then cover the (sometimes ephemeral) bash scripts and the deep embedding of regex in system administration tools like grep, find, chron and others that often act on files, filtering pipes, etc. Maybe more.

Any text editing application is likely to seem incomplete to most users without the ability to search content using regular expressions. A survey of over 2000 web developers by codeanywhere[1] indicates that the 10 tools in Table 2.1 are widely used. Support for features using regex is indicated there by checkmarks clean codeTools table and intro.

clean shellTools and intro Table 2.2.

clean sqlTools and intro Table 2.3.

2.1.1 Programming languages that support regex

For most popular programming languages, the ability to use regular expressions to search text is provided using standard libraries or is built into the language. Below is a list of standard regex libraries or built-ins *provided as a core language feature* for each of the top 20 most popular languages ordered according to the TIOBE[2] index on March 22, 2016:

1: Java java.util.regex

2: C ***NONE***

3: C++ std::regex

4: C# System.Text.RegularExpressions

5: Python re module

6: PHP PCRE core extension

[1] https://blog.codeanywhere.com/most-popular-ides-code-editors/
[2] http://www.tiobe.com/tiobe_index

Table 2.1: Regex-based feature breakdown for 10 popular code editing tools

Tool	Find	Replace	Feature3	Feature4	Feature 5	Feature 6
Notepad++	✓	✓	✓	✓	✓	
Sublime Text	✓	✓	×	×	×	
Eclipse	✓	✓	✓	×	×	
Netbeans	×	✓	×	✓	×	
IntelliJ	×	×	×	×	×	
Vim	✓	✓	✓	✓	✓	
Visual Studio	✓	✓	×	×	×	
PhpStorm	✓	✓	✓	×	×	
Atom	×	✓	×	✓	×	
Emacs	×	×	×	×	×	

✓= has feature, ×= does not have feature

Table 2.2: Regex-based feature descriptions for 7 popular command line tools

Tool	short description of regex usage
ls	short description of regex usage
find	short description of regex usage
grep	short description of regex usage
sed	short description of regex usage
awk	short description of regex usage
tool6	short description of regex usage
tool7	short description of regex usage

Table 2.3: Regex-based feature descriptions for 5 popular sql engines

Tool	short description of regex usage
Oracle	short description of regex usage
MySQL	short description of regex usage
MS SQL Server	short description of regex usage
MongoDB	short description of regex usage
PostgreSQL	short description of regex usage

7: Visual Basic .NET System.Text.RegularExpressions

8: JavaScript RegExp object (built-in)

9: Perl perlre core library

10: Ruby Regexp class (built-in)

11: Delpi RegularExpressions unit

12: Assembly language ***NONE***

13: Visual Basic ***NONE***

14: Swift NSRegularExpression

15: Objective-C NSRegularExpression

16: R grep (built-in)

17: Groovy java.util.regex

18: MATLAB regexp function (built-in)

19: PL/SQL LIKE operator (built-in)

20: D std.regex

Although pure ANSI C does not include a standard regex library or built-in, libraries providing regex support can be made available such as POSIX, PCRE or re2c. Similarly, pure Visual Basic has no core regex support but can use the RegExp object provided by the VBScript library. In fact, for most general-purpose languages, multiple alternative regex libraries can be found which may offer slightly different syntax or optimizations for speed. For example, the following libraries are alternatives to the C++ std::regex libary: Boost.Regex, Boost.Xpressive, cppre, DEELX, GRETA, Qt/QRegExp and RE2. These alternative libraries are developed by hobby users and software giants alike, with RE2 re2 (2015) being a recent and notable alternative library developed by Google.

The vast majority of modern regex libraries implement pattern syntax and feature sets based on PCRE standards with some exclusions or slightly different syntax for the same functionality. The major exception to this rule is SQL, which has it's own version of many features (underscore for characters, etc. SQL feature mini-table?). A complete analysis of the many subtle variations in syntax and implementation detail is beyond the scope of this thesis and is an opportunity for future work mentioned in the final discussion.

clean these thoughts So what are programming languages using regex for? It depends, but IMHO the capture group really shines in programming-language use, because captured content can be put into a variable and used later. Simple matching that requires the whole string to match seems less useful - unless we are validating user input. Note that regex are central to YACC and LEX, which are critical compiler tools for generating parsers used in the compilation process and lexing source files, respectively. So here regex are used as a meta-programming language

specifying the behavior of a parser. I use split all the time, usually splitting on a comma or tab, but this needs to be flexable, why not regex? This qualifies as worthwhile for future work.

2.2 Analyzing and testing regex

Regular expressions have been a focus point in a variety of research objectives. From the user perspective, tools have been developed to support more robust creation Spishak et al. (2012) or to allow visual debugging Beck et al. (2014). 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 Babbar and Singh (2010); Li et al. (2008).

Research tools like Hampi Kiezun et al. (2013), and Rex Veanes et al. (2010), and commercial tools like brics Møller (2010) all support the use of regular expressions in various ways. Hampi was developed in academia and uses regular expressions as a specification language for a constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in applications such as test case generation Anand et al. (2013); Tillmann et al. (2014). Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation.

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.[3] Others will automatically generate strings that are matched by the regular expessions Kiezun et al. (2013). Other tools will automatically generate regexes when given a list of strings to match Babbar and Singh (2010); Li et al. (2008). The commonality of such tools provides evidence that people need help with regex composition and understandability.

2.3 Composing Assistants

VerbalExpressions[4].

[3] https://regex101.com/
[4] https://github.com/VerbalExpressions/PHPVerbalExpressions

2.4 Special Applications for regex

Some data mining frameworks use regular expressions as queries (e.g., Begel et al. (2010)). Efforts have also been made to expedite the processing of regular expressions on large bodies of text Baeza-Yates and Gonnet (1996).

Regarding applications, regular expressions have been used for test case generation Ghosh et al. (2013); Galler and Aichernig (2014); Anand et al. (2013); Tillmann et al. (2014), and as specifications for string constraint solvers Trinh et al. (2014); Kiezun et al. (2013). Regexes are also employed in MySQL injection prevention Yeole and Meshram (2011) and network intrusion detection network (2015), or in more diverse applications like DNA sequencing alignment Arslan (2005) or querying RDF data Lee et al. (2010); Alkhateeb et al. (2009).

2.5 Formalisms and research addressing regex

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 Spishak et al. (2012). To aid in regex creation and understanding, tools have been developed to support more robust creation Spishak et al. (2012) or to allow visual debugging Beck et al. (2014). 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 Babbar and Singh (2010); Li et al. (2008).

2.6 Gap in fundamental research into regex use in practice

Although regex have provided an essential search functionality for software development for the last 47 years, are essential to parsing, compiling, security, database queries and user input validation, and are incorporated into all but the most low-level programming languages, no fundamental research has been published investigating user behaviors, preferences, use cases, pain points, or challenges in composition and comprehension. Faced with an open field, we formulated N questions to begin the work of filling this fundamental knowledge gap. The following section articulates the motivations behind the questions explored in this thesis.

2.7 Questions explored in this thesis and their motivations

2.7.1 RQ1: How are regex used in practice, especially what features are most commonly used?

Regex researchers and tool designers must pick what features to include or exclude, which can be a difficult design decision. Supporting advanced features may be more expensive, taking more time and potentially making the project too complex and cumbersome to execute well. A selection of only the simplest of regex features limits the applicability or relevance of that work. Despite extensive research effort in the area of regex support, no research has been done about how regexes are used in practice and what features are essential for the most common use cases.

2.7.2 RQ2: What behavioral categories can be observed in regex?

Clean these thoughts If we know some categories of regex behavior, then that gives good insight into what users are really doing with regex and in turn, what behaviors are most important for future regex technologies. Given a sample of the population of regexes in the wild, we expect to see some behavioral groups. But how to define behavior, and how to automate the investigation enough to handle a large number of regex? This analysis was very cpu-intensive and ran up against many implementation challenges, but is a successful first attempt to investigate regex composer's behaviors and needs.

2.7.3 RQ3: What preferences, behaviors and opinions do professional developers have about using regex?

Clean these thoughts Why not just ask software developers about their use habits and preferences in a survey? That's what we did here. But it's important to mention that these questions had the benefit of the feature and behavioral analysis

2.7.4 RQ4: Within five equivalence classes, what representations are most frequently observed?

Clean these thoughtsThere are many ways to represent the same functional regex, that is, the user has choices to make about how to compose a regex for any given task. Assuming that regex composers will tend to choose the best representation most of the time, we want to know what representation choices are most frequent.

2.7.5 RQ5: What representations are more comprehensible?

Clean these thoughts After defining the equivalence classes and potential regex refactorings 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, understandable, and performance. We focus on refactoring for understandability.

2.7.6 RQ6: For each equivalence class, which representation is preferred according to frequency and comprehensibility?

Clean these thoughtsThis section formalizes a technique of ordering the data from the previous two sections.

2.8 Surveys of regex research

clean these thoughts We've got a handful of surveys that have been done exploring the state of the art in regex: Brzozowski in 1962 did the first survey of applications,

2.9 Mining

Exploring language feature usage by mining source code has been studied extensively for Smalltalk Callaú et al. (2011), JavaScript Richards et al. (2010), and Java Dyer et al. (2014); Grechanik et al. (2010); Parnin et al. (2013); Livshits et al. (2005), and more specifically, Java generics Parnin et al. (2013) and Java reflection Livshits et al. (2005). Our prior work (Chapman and Stolee (2016), under review) was the first to mine and evaluate regular expression

usages from existing software repositories. The intention of the prior work Chapman and Stolee (2016) 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.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns Linares-Vásquez et al. (2014) and bug characterizations Chen et al. (2014). Exploring language feature usage by mining source code has been studied extensively for Smalltalk Callaú et al. (2011, 2013), JavaScript Richards et al. (2010), and Java Dyer et al. (2014); Grechanik et al. (2010); Parnin et al. (2013); Livshits et al. (2005), and more specifically, Java generics Parnin et al. (2013) and Java reflection Livshits et al. (2005). To our knowledge, this is the first work to mine and evaluate regular expression usages from existing software repositories. Related to mining work, regular expressions have been used to form queries in mining framework Begel et al. (2010), but have not been the focus of the mining activities. Surveys have been used to measure adoption of various programming languages Meyerovich and Rabkin (2013); Dattero and Galup (2004), and been combined with repository analysis Meyerovich and Rabkin (2013), but have not focused on regexes.

2.10 Refactoring and smells

Regular expression refactoring has also not been studied directly, though refactoring literature abounds Mens and Tourwé (2004); Opdyke (1992); Griswold and Notkin (1993). The closest to regex refactoring comes from research toward expediting the processing of regular expressions on large bodies of text Baeza-Yates and Gonnet (1996), which could be thought of as refactoring for performance.

In software, code smells have been found to hinder understandability of source code Abbes et al. (2011); Du Bois et al. (2006). 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.

Code smells in object-oriented languages were introduced by Fowler Fowler (1999). Researchers have studied the impact of code smells on program comprehension Abbes et al. (2011); Du Bois et al. (2006), 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 Hermans et al. (2012, 2014); Stolee and Elbaum (2011, 2013). 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 Stolee and Elbaum (2011, 2013).

CHAPTER 3. Feature Analysis

3.1 Utilizations of the re module

Utilization: A *utilization* occurs whenever a regex appears in source code. We detect utilizations by statically analyzing source code and recording calls to the `re` module in Python. Within a source code file, a utilization is composed of a function, a pattern, and 0 or more flags. Figure 3.1 presents an example of one regex utilization, with key components labeled. The function call is `re.compile`, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. When executed, this utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. Thought of another way, a regex utilization is one single invocation of the `re` library.

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 3.1. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 3.1 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 `-`).

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)

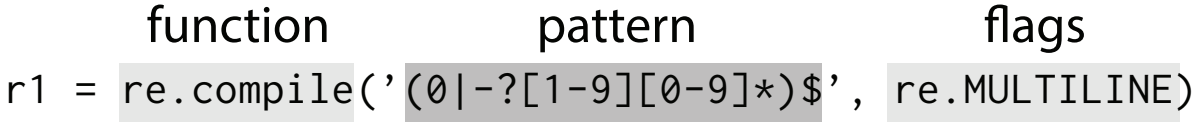

Figure 3.1: Example of one regex utilization

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.

In this work, we primarily focus on patterns since they are cross-cutting across languages and are the primary way of specifying the matching behavior. Next, we describe the research questions, data set collection and analysis.

We consider regex language features to be tokens that specify the matching behavior of a regex pattern, for example, the `+` in `ab+`. All studied features are listed and described in Table 3.1 with examples. We then map the feature coverage for four common regex support tools, brics, hampi, RE2 and Rex, and explore survey responses regarding feature usage for some of the less supported features.

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. For example, the pattern `` `ab*c' `` will match strings beginning with a, followed by zero or more b's, and ending in c, such as: 'ac'!, 'abc', 'abbc', etc.

Regexes are commonly used for parsing text using a general purpose language like Python, validating content entered into web forms using JavaScript, and searching text files for a particular pattern using tools like grep, vim or Eclipse.

Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports Spishak et al. (2012).

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 Yeole and Meshram (2011) and network intrusion detection network (2015). However, recent research has suggested that regular expressions (regexes) are hard to understand, hard to compose, and error prone Spishak et al. (2012). 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.

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 3,898 projects.

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 pre-processed by removing Python quotes (`'\\W'` becomes `\\W`), unescaping escaped characters (`\\W` becomes `\W`) and parsing the resulting string using an ANTLR-based, open source PCRE parser[1]. 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.

[1] https://github.com/bkiers/pcre-parser

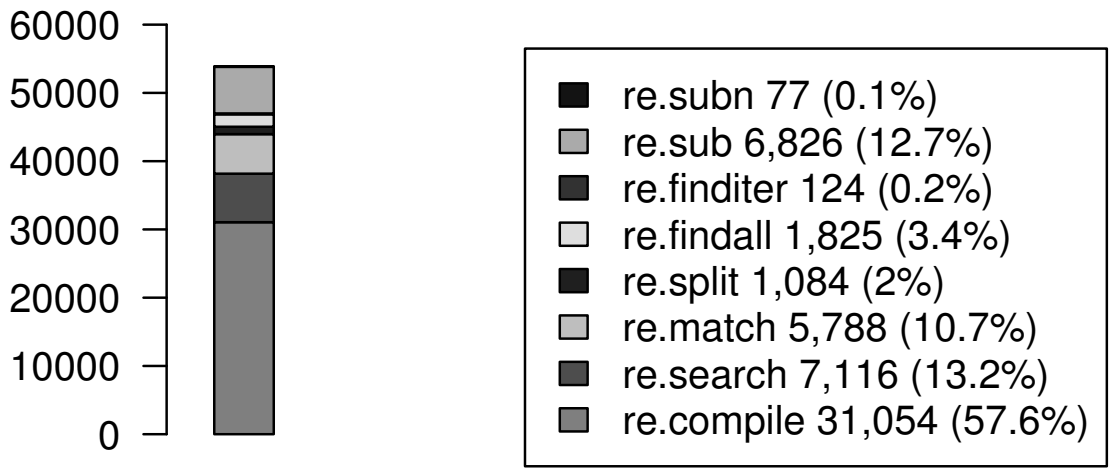

Figure 3.2: How often are `re` functions used?

The 13,597 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We refer to this set of weighted, distinct pattern strings as the *corpus*.

Third, we investigate what features are supported by four large projects that aim to support regex usage (brics Møller (2010), hampi Kiezun et al. (2013), Rex Veanes et al. (2010), and RE2 re2 (2015)), and which features are not supported, but are frequently used by developers.

We explore regex utilizations and flags used in the scraped Python projects. Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. To illustrate how saturated projects are with regexes, we measure utilizations per project, files scanned per project, files contained utilizations, and utilizations per file, as shown in Table 5.3.

Of projects containing at least one utilization, the average utilizations per project was 32 and the maximum was 1,427. The project with the most utilizations is a C# project[2] that maintains a collection of source code for 20 Python libraries, including larger libraries like `pip`, `celery` and `ipython`. These larger Python libraries contain many utilizations. From Table 5.3, we also see that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations.

[2] https://github.com/Ouroboros/Arianrhod

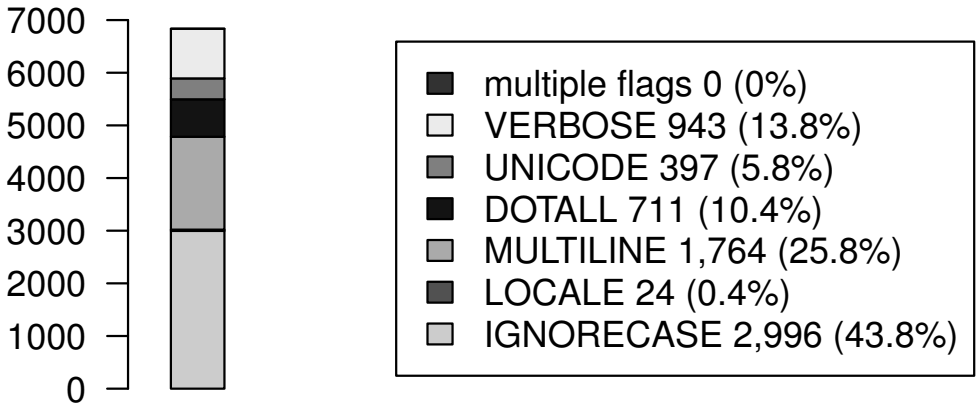

Figure 3.3: Which behavioral flags are used?

Figure 3.4: Two patterns parsed into feature vectors

3.2 Building the corpus of patterns

For each escaped pattern, the PCRE-parser produces a tree of feature tokens, which is converted to a vector by counting the number of each token in the tree. For a simple example, consider the patterns in Figure 3.4. The pattern `` `^m+(f(z)*)+' `` contains four different types of tokens. It has the kleene star (KLE), which is specified using the asterisk `` `*' `` character, additional repetition (ADD), which is specified using the plus `` `+' `` character, capture groups (CG), which are specified using pairs of parenthesis `` `(...)' `` characters, and the start anchor (STR), which is specified using the caret `` `^' `` character at the beginning of a pattern. A list of all features and abbreviations is provided in Table 3.1.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking the number of patterns and projects that the each feature appears in at least once.

3.2.0.1 Feature Usage

Table 3.1 displays feature usage from the corpus and relates it to four major regex related projects. Only features appearing in at least 10 projects are listed. The first column, *rank*,

lists the rank of a feature (relative to other features) in terms of the number of projects in which it appears. The next column, *code*, gives a succinct reference string for the feature, and is followed by a *description* column that provides a brief comment on what the feature does. The *example* column provides a short example of how the feature can be used.

The next four columns, (i.e., *brics*, *hampi*, *Rex*, and *RE2*), map to the four major research projects chosen for our investigation. We indicate that a project supports a feature with the '●' symbol, and indicate that a project does not support the feature with the '○' symbol. The final four columns contain two pairs of usage statistics. The first pair contains the number and percent of *patterns* that a feature appears in, out of the 13,597 patterns that make up the corpus. The second pair of columns contain the number and percent of *projects* that a feature appears in out of the 1,645 projects scanned that contain at least one utilization.

One notable omission from Table 3.1 is the literal feature, which is used to specify matching any specific character. An example pattern that contains only one literal token is the pattern `` `a' ``. This pattern only matches the lowercase letter 'a'. The literal feature was found in 97.7% of patterns.

We consider the literal feature to be necessary for any regex related tool to support, and so exclude it from Table 3.1 and the rest of the feature analysis.

The eight most commonly used features, ADD, CG, KLE, CCC, ANY, RNG, STR and END, appear in over half the projects. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%), even though they appear in similar numbers of projects.

3.2.0.2 Feature Support in Regex Tools

While there are many regex tools available, in this work, we focus on the feature support for four tools, brics, hampi, Rex and RE2, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and applications. Further, as we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

To create the tool mappings, we consulted documentation for each tool. For brics, we collected the set of supported features using the formal grammar[3]. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository[4] (this may have been an overestimation, as this included more features than specified by the formal grammar[5]). For RE2, we used the supported feature documentation[6]. For Rex, we collected the feature set empirically because we tried to parse all scraped patterns with Rex for the behavioral analysis , and Rex provides comprehensive error feedback for unsupported features.

Of the four projects selected for this analysis, RE2 supports the most studied features (28 features) followed by hampi (25 features), Rex (21 features), and brics (12 features). All projects support the 8 most commonly used features except brics, which does not support STR or END.

No projects support the four look-around features LKA, NLKA, LKB and NLKB. RE2 and hampi support the LZY, NCG, PNG and OPT features, whereas brics and Rex do not.

3.3 Discussion of utilization and feature analysis results

3.3.1 Implications

Think about the part of all regex that use any back-references and so are not representing regular languages (vs those that are).

3.3.2 Opportunities for future work

3.3.3 Threats to validity

[3] http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html
[4] https://code.google.com/p/hampi/downloads/list
[5] http://people.csail.mit.edu/akiezun/hampi/Grammar.html
[6] https://re2.googlecode.com/hg/doc/syntax.html

Table 3.1: How frequently do features appear in projects?

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nProjects
1	ADD	one-or-more repetition	`z+`	●	●	●	●	6,003	44.1	1,204
2	CG	a capture group	`(caught)`	●	●	●	●	7,130	52.4	1,194
3	KLE	zero-or-more repetition	`.*`	●	●	●	●	6,017	44.3	1,099
4	CCC	custom character class	`[aeiou]`	●	●	●	●	4,468	32.9	1,026
5	ANY	any non-newline char	`.`	●	●	●	●	4,657	34.3	1,005
6	RNG	chars within a range	`[a-z]`	●	●	●	●	2,631	19.3	848
7	STR	start-of-line	`^`	○	●	●	●	3,563	26.2	846
8	END	end-of-line	`$`	○	●	●	●	3,169	23.3	827
9	NCCC	negated CCC	`[^qwxf]`	●	●	●	●	1,935	14.2	776
10	WSP	\t \n \r \v \f or space	`\s`	○	●	●	●	2,846	20.9	762
11	OR	logical or	`a|b`	●	●	●	●	2,102	15.5	708
12	DEC	any of: 0123456789	`\d`	○	●	●	●	2,297	16.9	692
13	WRD	[a-zA-Z0-9_]	`\w`	○	●	●	●	1,430	10.5	650
14	QST	zero-or-one repetition	`z?`	●	●	●	●	1,871	13.8	645
15	LZY	as few reps as possible	`z+?`	○	●	○	●	1,300	9.6	605
16	NCG	group without capturing	`a(?:b)c`	○	●	○	●	791	5.8	404
17	PNG	named capture group	`(?P<name>x)`	○	●	○	●	915	6.7	354
18	SNG	exactly n repetition	`z{8}`	●	●	●	●	581	4.3	340
19	NWSP	any non-whitespace	`\S`	○	●	●	●	484	3.6	270
20	DBB	$n \leq x \leq m$ repetition	`z{3,8}`	●	●	●	●	367	2.7	238
21	NLKA	sequence doesn't follow	`a(?!yz)`	○	○	○	○	131	1	183
22	WNW	word/non-word boundary	`\b`	○	○	○	●	248	1.8	166
23	NWRD	non-word chars	`\W`	○	●	●	●	94	0.7	165
24	LWB	at least n repetition	`z{15,}`	●	●	●	●	91	0.7	158
25	LKA	matching sequence follows	`a(?=bc)`	○	○	○	○	112	0.8	158
26	OPT	options wrapper	`(?i)CasE`	○	●	○	●	231	1.7	154
27	NLKB	sequence doesn't precede	`(?<!x)yz`	○	○	○	○	94	0.7	137
28	LKB	matching sequence precedes	`(?<=a)bc`	○	○	○	○	80	0.6	120
29	ENDZ	absolute end of string	`\Z`	○	○	○	●	89	0.7	90
30	BKR	match the i^{th} CG	`\1`	○	○	○	○	60	0.4	84
31	NDEC	any non-decimal	`\D`	○	●	●	●	36	0.3	58
32	BKRN	references PNG	`\g<name>`	○	●	○	○	17	0.1	28
33	VWSP	matches U+000B	`\v`	○	○	●	●	13	0.1	15
34	NWNW	negated WNW	`\B`	○	○	○	●	4	0	11

CHAPTER 4. Behavioral clustering

4.1 Experimental design

4.1.1 Conceptual basis

An ideal analysis of regex behavioral similarity would use subsumption or containment analysis. However, we struggled to find a tool that could facilitate such an analysis. Further, regular expressions in code libraries (e.g., for Python, Java) are not the same as regular languages in formal language theory. Some features of regular expression libraries, such as backreferences, make the libraries more expressive than regular languages. This allows a regular expression pattern to match, for example, repeat words, such as "cabcab", using the pattern `([a-z]+)\1`. However, building an automaton to recognize such a pattern and to facilitate containment analysis, is infeasible. For these reasons, we developed a similarity analysis based on string matching.

4.1.2 Overview of process

Our similarity analysis clusters regular expressions by their behavioral similarity on matched strings. Consider two unspecified patterns `A` and `B`, a set `mA` of 100 strings that pattern `A` matches, and a set `mB` of 100 strings that pattern `B` matches. If pattern `B` matches 90 of the 100 strings in the set `mA`, then `B` is 90% similar to `A`. If pattern `A` only matches 50 of the strings in `mB`, then `A` is 50% similar to `B`. We use similarity scores to create a similarity matrix as shown

```
Pattern A matches 100/100 of A's strings
Pattern B matches  90/100 of A's strings
Pattern A matches  50/100 of B's strings
Pattern B matches 100/100 of B's strings
```

	A	B
A	1.0	0.9
B	0.5	1.0

Figure 4.1: A similarity matrix created by counting strings matched

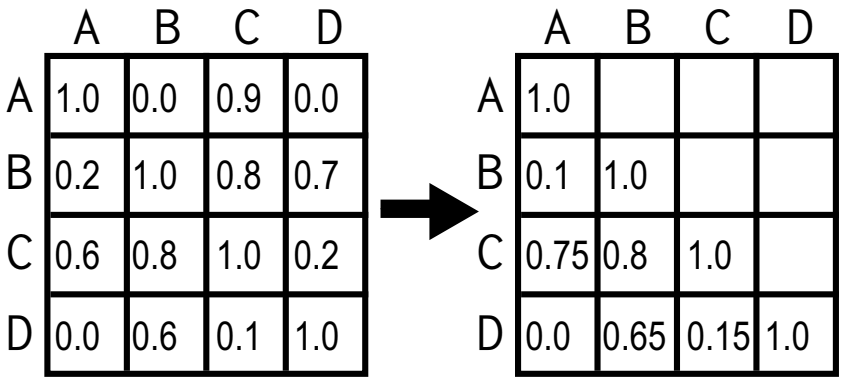

Figure 4.2: Creating a similarity graph from a similarity matrix

in Figure 4.1. In row A, column B we see that B is 90% similar to A. In row B, column A, we see that A is 50% similar to B. Each pattern is always 100% similar to itself, by definition.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix are averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 4.2. This facilitates clustering using the Markov Clustering (MCL) algorithm[1]. We chose MCL because it offers a fast and tunable way to cluster items by similarity and it is particularly useful when the number of clusters is not known *a priori*.

In the implementation, strings are generated for each pattern using Rex Veanes et al. (2010). Rex generates matching strings by representing the regular expression as an automaton, and then passing that automation to a constraint solver that generates members for it[2]. If the regex matches a finite set of strings smaller than 400, Rex will produce a list of all possible strings. Our goal is to generate 400 strings for each pattern to balance the runtime of the similarity analysis with the precision of the similarity calculations.

For clustering, we prune the similarity matrix to retain all similarity values greater than or equal to 0.75, setting the rest to zero, and then using MCL. This threshold was selected based on recommendations in the MCL manual. The impact of lowering the threshold would likely result in either the same number of more diverse clusters, or a larger number of clusters, but is unlikely to markedly change the largest clusters or their summaries, which are the focus of our analysis for some research question reference. , but further study is needed to substantiate this claim. We also note that MCL can also be tuned using many parameters, including inflation and

[1] http://micans.org/mcl/
[2] http://research.microsoft.com/en-us/projects/rex/

filtering out all but the top-k edges for each node. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and k=83. The top 100 clusters are categorized by inspection into six categories of behavior.

The end result is clusters and categories of highly behaviorally similar regular expressions, though we note that this approach has a tendency to over-approximate the similarity of two regexes. We measure similarity based on a finite set of generated strings, but some regexes match an infinite set (e.g., `ab*c`), so measuring similarity based on the first 400 strings may lead to an artificially high similarity value. To mitigate this threat, we chose a large number of generated strings for each regex, but future work includes exploring other approaches to computing regex similarity.

4.2 Similarity matrix creation

4.2.1 Implementation details

In clustering the regular expressions, we are most interested in observing behavior of regexes found in multiple projects. Starting with the 13,597 patterns of the corpus, we discarded 10,015 (74%) patterns that were not found in multiple projects. Then we excluded an additional 711 (5%) patterns that contain features not supported by Rex. We studied the remaining 2,871 (21%) patterns using our similarity analysis technique. The impact is that 923 projects were excluded from the data set for the similarity analysis. Omitted features are indicated in Table 3.1 for Rex.

Table 4.1: Sample from an example cluster

index	pattern	nProjects	index	pattern	nProjects
1	`:+'	8	5	`[:]'	6
2	`(:)'	8	6	`([^:]+):(.*)'	6
3	`(:+)'	8	7	`\s*:\s*'	4
4	`(:)(:*)'	8	8	`\:'	2

4.2.2 Results

4.3 Markov clustering

4.3.1 Background

4.3.2 Tuning parameters

4.3.3 Results

From 2,871 distinct patterns, MCL clustering identified 186 clusters with 2 or more patterns, and 2,042 clusters of size 1. The average size of clusters larger than size one was 4.5. Each pattern belongs to exactly one cluster.

Three example strings generated by Rex for the first pattern are: '-()', '*'8(5)', 'Oe()'. For the third pattern, Rex generated these three strings: ' ()', '(q)F', '(n)M'. The pattern: `\(.*\)$` is very similar, but will not match the string '(n)M', and so was placed in a different cluster.

Table 4.1 provides an example of a behavioral cluster containing 12 patterns (four longer patterns omitted for brevity). Patterns from this cluster are present in 31 different projects. All patterns in this cluster share the literal ':' character. The smallest pattern, `` `:+' ``, matches one or more colons.

Another pattern from this cluster, `([^:]+):(.*)`, requires at least one non-colon character to occur before a colon character. Our similarity value between these two regexes was below the minimum of 0.75 because Rex generated many strings for ':+' that start with one or more colons. We observe that the smallest pattern in a cluster provides insight about key characteristic that all the patterns in the cluster have in common. A shorter pattern will tend

to have less extraneous behavior because it is specifying less behavior, yet, in order for the smallest pattern to be clustered, it had to match most of the strings created by Rex from many other patterns within the cluster, and so we observe that the smallest pattern is useful as a representative of the cluster.

For the rest of this paper, a cluster will be represented by one of the shortest patterns it contains, followed by the number of projects any member of the cluster appears in, so the cluster in Table 4.1 will be represented as `` `:+'(31) ``. This representation is not an attempt to express all notable behavior of patterns within a cluster, but is a useful and meaningful abbreviation. Other regexes in the cluster may exhibit more diverse behavior, for example the pattern `` `([^: ]+):(.*)' `` requires a non-colon character to appear before a colon character.

4.4 Categorization of clusters

4.4.1 Implementation details

4.4.2 Results

We manually mapped the top 100 largest clusters based on the number of projects into 6 behavioral categories (determined by inspection). The largest cluster was left out, as it was composed of patterns that trivially matched almost any string, like `` `b*' `` and `` `^' ``. The remaining 99 clusters were all categorized. These clusters are briefly summarized in Table 4.2, showing the name of the category and the number of clusters it represents, patterns in those clusters, and projects. The most common category is *Multi Matches*, which contains clusters that have alternate behaviors (e.g., matching a comma or a semicolon, as in `` `,|;'(18) ``). Each cluster was mapped to exactly one category. Next, we describe the categories, ordered by the number of projects the regex patterns map to.

4.4.2.1 Multiple Matching Alternatives

The patterns in these clusters match under a variety of conditions by using a character class or a disjunctive `|`. For example: `` `(\W)'(89) `` matches any alphanumeric character, `` `(\s)'(89) `` matches any whitespace character, `` `\d'(58) `` matches any numeric character, and `` `,|;'(18) ``

Table 4.2: Cluster categories and sizes (RQ4)

Category	Clusters	Patterns	Projects
Multi Matches	21	237	295
Specific Char	17	103	184
Anchored Patterns	20	85	141
Content of Parens	10	46	111
Two or More Chars	16	40	120
Code Search	15	27	92

matches a comma or semicolon. Most of these clusters are represented by patterns that use default character classes, as opposed to custom character classes. This provides further support for our survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which a majority of participants indicated they use the default classes more than custom. This category contains 21 clusters, each appearing in an average of 33 projects.

4.4.2.2 Specific Character Must Match

Each cluster in this category requires one specific character to match, for example: `` `\n\s*'(42) `` matches only if a newline is found, `` `:+'(31) `` matches only if a colon is found, `` `%'(22), `` matches only if a percent sign is found and `` `}'(14) `` matches only if a right curly brace is found. The commonality of this cluster category contrasts with the survey in (Section) in which participants reported to very rarely or never use regexes to check for a single character (Table 5.2). This category contains 17 clusters, each appearing in an average of 17.1 projects. These clusters have a combined total of 103 patterns, with at least one pattern present in 184 projects.

4.4.2.3 Anchored Patterns

Each of the clusters uses at least one endpoint anchor to require matches to be absolutely positioned, for example: `` `(\w+)$'(35) `` captures the word characters at the end of the input, `` `^\s'(16) `` matches a whitespace at the beginning of the input, and `` `^-?\d+$'(17) `` requires

that the entire input is an (optionally negative) integer. These anchors are the only way in regexes to guarantee that a character does (or does not) appear at a particular location by specifying what is allowed. As an example, `^[-_A-Za-z0-9]+$` says that from beginning to end, only `[-_A-Za-z0-9]` characters are allowed, so it will fail to match if undesirable characters, such as ?, appear anywhere in the string. This category contains 20 clusters, each appearing in an average of 15.4 projects. These clusters have a combined total of 85 patterns, with at least one pattern present in 141 projects.

The thing I want to mention about anchored patterns (but have struggled to say in the past) is that they are the only way to guarantee that a character does not appear in a particular location by specifying what is allowed. Consider the regex `^[-_A-Za-z0-9]+$` which will fail to match if an undesirable character like '?' appears anywhere in the input. In logic, there is a similar phenomenon. That is, 'Always' is true iff 'Not Exists' of the negation is true, and by requiring an entire input to always maintain some abstraction, you can indirectly specify the negation of another (inverse) abstraction. Even with only one anchor point, a regex like `.*[0-9]$` is creating an ultimatum about the end being a digit. Without the endpoint anchors, I don't see how one could specify absolutes about an input.

4.4.2.4 Content of Brackets and Parenthesis

The clusters in this category center around finding a pair of characters that surround content, often also capturing that content. For example, `` `\(.*\)'(29) `` matches when content is surrounded by parentheses and `` `".*"'(25) `` matches when content is surrounded by double quotes. The cluster `` `<(.+)>'(23) `` matches and captures content surrounded by angled brackets. This category contains 10 clusters, each appearing in an average of 18.4 projects. These clusters have a combined total of 46 patterns, with at least one pattern present in 111 projects.

include this?, and `` `\[.*\]'(22) `` matches when content is surrounded by square brackets

4.4.2.5 Two or More Characters in Sequence

These clusters require several characters in a row to match some pattern, for example: `` `\d+\.\d+'(30) `` requires one or more digits followed by a period character, followed by one or more digits. The cluster `` `  '(17) `` requires two spaces in a row, `` `([A-Z][a-z]+[A-Z][^ ]+)'(11) ``, and `` `@[a-z]+'(9) `` requires the at symbol followed by two or more lowercase characters, as in a

twitter handle. This category contains 16 clusters, each appearing in an average of 13 projects. These clusters have a combined total of 40 patterns, with at least one pattern present in 120 projects.

Again, it might be interesting to look at what particular sequences are looking like. I think I mention this again in the discussion, but should we put it here instead?

4.4.2.6 Code Search and Variable Capturing

These clusters show a recognizable effort to parse source code or URLs. For example, `` `^https?://'(23) `` matches a web address, and `` `(.+)=(.+)'(9) `` matches an assignment statement, capturing both the variable name and value. The cluster `` `\$\{([\w\-]+)\}'(11) `` matches an evaluated string interpolation and captures the code to evaluate. This category contains 15 clusters, each appearing in an average of 11.7 projects. These clusters have a combined total of 27 patterns, with at least one pattern present in 92 projects.

4.5 Discussion of cluster categories

4.5.1 Implications

When tool designers are considering what features to include, data about usage in practice is valuable. Behavioral similarity clustering helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in matching behavior. We are also able to find out what features are being used in these behavioral trends so that we can make assertions about why certain features are important. We used the behavior of individual patterns to form clusters, and identified six main categories for the clusters. Overall, we see that many clusters are defined by the presence of particular tokens, such as the colon for the cluster in Table 4.1. We identified six main categories that define regex behavior at a high level: matching with alternatives, matching literal characters, matching with sequences, matching with endpoint anchors, parsing contents of brackets or braces, or searching and capturing code, and can be considered in conjunction with the self-described regex activities from the survey in Table 5.2 to be representative of common uses for regexes. One of the six common cluster

categories, *Code Search and Variable Capturing*, has a very specific purpose of parsing source code files. This shows a very specific and common use of regular expressions in practice.

4.5.1.1 Finding Specific Content

Two categorical clusters, *Specific Characters Must Match* and *Two or More Characters in Sequence*, deal with identifying the presence of specific character(s). While multiple character matching subsumes single character matching, the overarching theme is that these regexes are looking to validate strings based on the presence of very specific content, as would be done for many common activities listed in Table 5.2, such as, "Locating content within a file or files." More study is needed into what content is most frequently searched for, but from our cluster analysis we found that version numbers, twitter or user handles, hex values, decimal numbers, capitalized words, and particular combinations of whitespace, slashes and other delimiters were discernible targets.

Capturing the contents of brackets and searching for delimiter characters were some of the most apparent behavioral themes observed in our regex clusters, and developers frequently use regexes to parse source code.

clean these thoughts Previous discussions have been compacted until nearly meaningless, but here is what people are really doing, as revealed by the clustering:

- parse a line of source code to capture an assignment using '='
- capture/find/identify/count identifiers like emails, usernames, twitter handles, etc
- parse some structured file, maybe xml or maybe just something home-rolled that is regular by design
- similarly, finding a special marker that you are expecting. For example, if you write a program that inserts a colon between some fields when serializing an object, then you can de-serialize using the colon. There is so much freedom that it's a hard pattern to detect, but I think this is where the single-character focus comes in the most often.
- foolishly parse contents of brackets. This can be okay in certain controlled contexts.

- similarly foolishly parsing contents of double-quotes.
- breaking down a log file, maybe counting presence/absence of something or summing a field when present
- parsing a simple string code like a date format, certainly html escape codes
- parsing a phone number, ssn, numeric date, regular number
- absolutely parsing urls. period. mostly this. Also IP addresses. All sorts of web protocols are regular.
- scanning for keywords using an OR. This is dubious when it's like password—secret—hash, so someone is fishing...
- scanning for an expected error message prefix like 'Invalid object specification:'
- scanning for objects and function calls like 'prefs.add.∗'
- scanning for HTML content like 'a href='
- really generic stuff like numbers and then some space...so vanilla it is hard to say what they were doing without going back to their code.

4.5.2 Opportunities for future work

4.5.3 Threats to validity

CHAPTER 5. Developer Survey

5.1 Survey design based on feature analysis

To understand the context of when and how programmers use regular expressions, we designed a survey, implemented using Google Forms, with 40 questions. The questions asked about regex usage frequency, languages, purposes, pain points, and the use of various language features. [1] Participation was voluntary and participants were entered in a lottery for a $50 gift card.

Our goal was to understand the practices of professional developers. Thus, we deployed the survey to 22 professional developers at Dwolla, a small software company that provides tools for online and mobile payment management. While this sample comes from a single company, we note anecdotally that Dwolla is a start-up and most of the developers worked previously for other software companies, and thus bring their past experiences with them. Surveyed developers have nine years of experience, on average, indicating the results may generalize beyond a single, small software company, but further study is needed.

5.2 Summary of survey results

The survey was completed by 18 participants (82% response rate) that identified as software developer/maintainers. Respondents have an average of nine years of programming experience ($\sigma = 4.28$). On average, survey participants report to compose 172 regexes per year ($\sigma = 250$) and compose regexes on average once per month, with 28% composing multiple regexes in a week and an additional 22% composing regexes once per week. That is, 50% of respondents uses regexes at least weekly. Table 5.1 shows how frequently participants compose regexes using

[1] https://github.com/softwarekitty/tour_de_source/blob/master/regex_usage_in_practice_survey.pdf

Table 5.1: Survey results for number of regexes composed per year by technical environment

Language/Environment	0	1-5	6-10	11-20	21-50	51+
General (e.g., Java)	1	6	5	3	1	2
Scripting (e.g., Perl)	5	4	3	3	2	1
Query (e.g., SQL)	15	2	0	0	1	0
Command line (e.g., grep)	2	5	3	2	0	6
Text editor (e.g., IntelliJ)	2	5	0	5	1	5

Table 5.2: Survey results for regex usage frequencies for activities, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

Activity	**Frequency**
Locating content within a file or files	4.4
Capturing parts of strings	4.3
Parsing user input	4.0
Counting lines that match a pattern	3.2
Counting substrings that match a pattern	3.2
Parsing generated text	3.0
Filtering collections (lists, tables, etc.)	3.0
Checking for a single character	1.7

each of several languages and technical environments. Six (33%) of the survey participants report to compose regexes using general purpose programming languages (e.g., Java, C, C#) 1-5 times per year and five (28%) do this 6-10 times per year. For command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year. Yet, regexes were rarely used in query languages like SQL. Upon further investigation, it turns out the surveyed developers were not on teams that dealt heavily with a database.

Table 5.2 shows how frequently, on average, the participants use regexes for various activities. Participants answered questions using a 6-point likert scale including very frequently (6), frequently (5), occasionally (4), rarely (3), very rarely (2), and never (1). Averaging across par-

Table 5.3: How saturated are projects with utilizations?

source	Q1	Avg	Med	Q3	Max
utilizations per project	2	32	5	19	1,427
files per project	2	53	6	21	5,963
utilizing files per project	1	11	2	6	541
utilizations per file	1	2	1	3	207

ticipants, among the most common usages are capturing parts of a string and locating content within a file, with both occurring somewhere between occasionally and frequently.

Using a similar 7-point likert scale that includes 'always' as a seventh point, developers indicated that they test their regexes with the same frequency as they test their code (average response was 5.2, which is between frequently and very frequently). Half of the developers indicate that they use external tools to test their regexes, and the other half indicated that they only use tests that they write themselves. Of the nine developers using tools, six mentioned online composition aides such as `regex101.com` where a regex and input string are entered, and the input string is highlighted according to what is matched.

When asked an open ended question about pain points encountered with regular expressions, we observed three main categories. The most common, "hard to compose," was represented in 61% (11) responses. Next, 39% (7) developers responded that regexes are "hard to read" and 17% (3) indicated difficulties with "inconsistency across implementations," which manifest when using regexes in multiple languages. These responses do not sum to 18 as three developers provided multiple parts in their answers.

Common uses of regexes include locating content within a file, capturing parts of strings, and parsing user input. The fact that all the surveyed developers compose regexes, and half of the developers use tools to test their regexes indicates the importance of tool development for regex. Developers complain about regexes being hard to read and hard to write.

The pattern language for Python, which is used to compose regexes, supports default character classes like the ANY or dot character class: `.` meaning, 'any character except newline'.

Table 5.4: Survey results for preferences between custom character and default character classes

Preference	Frequency
use only CCC	1
use CCC more than default	5
use both equally	2
use default more than CCC	10
use only default	2

It also supports three other default character classes: `\d`, `\w`, `\s` (and their negations). All of these default character classes can be simulated using the custom character class (CCC) feature, which can create semantically equivalent regexes. For example the decimal character class: `\d` is equivalent to a CCC containing all 10 digits: `\d` ≡ `[0123456789]` ≡ `[0-9]`.

Other default character classes such as the word character class: `\w` may not be as intuitive to encode in a CCC: `[a-zA-Z0-9_]`.

Survey participants were asked if they use only CCC, use CCC more than default, use both equally, use default more than CCC or use only default. Results for this question are shown in Table 5.4, with 67% (12) indicating that they use default the most.

Participants who favored CCC indicated that "it is more explicit," whereas the participants who favored default character classes said, "it is less verbose" and "I like using built-in code."

To further explore how participants use various regex features, participants were asked five questions about how frequently they use specific related groups of features:

- endpoint anchors (STR, END): `^` and `$`
- capture groups(CG): (capture me)
- word boundaries (WNW): `word\b`
- (negative) look-ahead/behinds (LKA, NLKA, LKB, NLKB): `a(?=bc)`, `(?<x)yz!`, `(?<=a)`, `a(?yz)!`
- lazy repetition (LZY): `ab+?`, `xy{2,3}?`

Table 5.5: Survey results for regex usage frequencies, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

Group	**Code**	**Frequency**
endpoint anchors	(STR, END)	4.4
capture groups	(CG)	4.2
word boundaries	(WNW)	3.5
lazy repetition	(LZY)	2.9
(neg) look-ahead/behind	(LKA, NLKA, LKB, NLKB)	2.5

Results are shown in Table 5.5, indicating that lazy repetition and look-ahead features are rarely used and capture groups and endpoint anchors are occasionally to frequently used.

Our results indicate that regexes are most frequently used in command line tools and IDEs.

5.3 Discussion of survey results

5.3.1 Implications

5.3.2 Opportunities for future work

5.3.3 Threats to validity

CHAPTER 6. Equivalent representations of regex and their frequencies in the corpus

6.1 Experiment design

6.1.1 Defining five equivalence classes

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 preferred representations for four of the five equivalence classes based on mutual agreement between community standards and understandability, with three of those being statistically significant. 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 preferred regex representations and refactorings 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 as well as our two empirical studies, one using source code artifacts and another using human participants.

After studying over 13,000 distinct regex strings from nearly 4,000 Python projects, 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 6.1 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

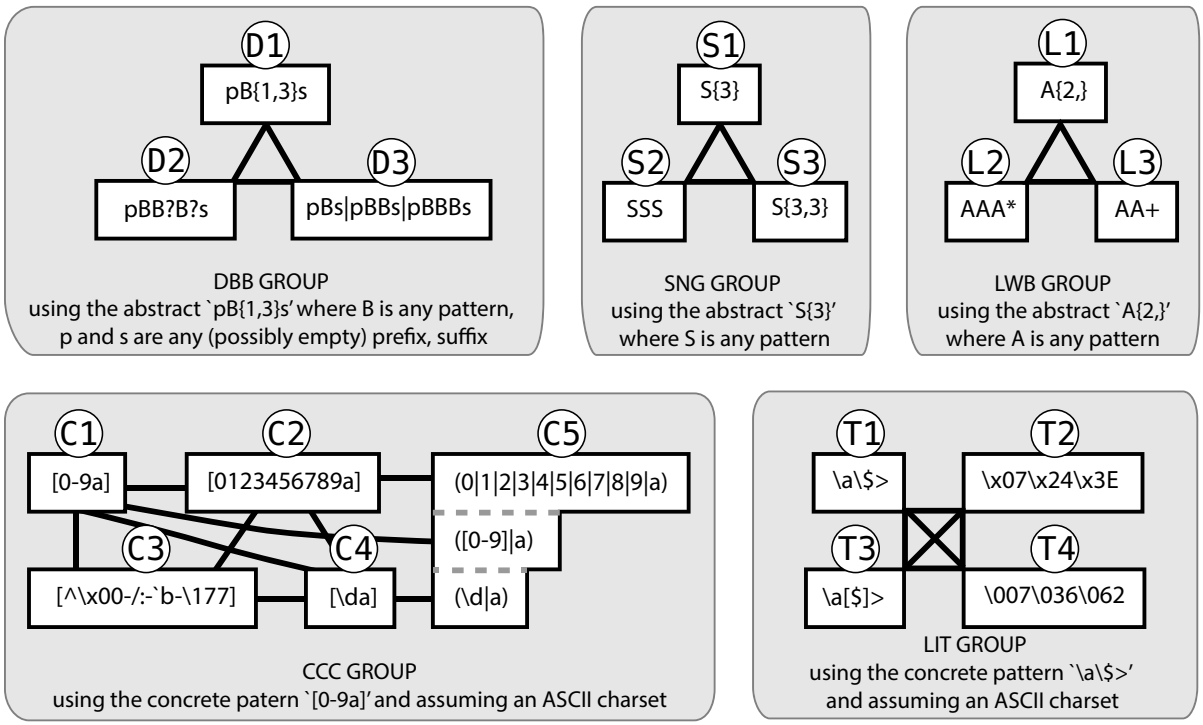

Figure 6.1: Equivalence classes with various representations of semantically equivalent refactorings within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

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:

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. For example, the pattern `[^ao]` matches every character *except* `a` or `o`. If the applicable character set is known (e.g., ASCII, UTF-8, etc.), then any non-negative character class can be represented as a negative character class. For example, assuming an ASCII charset that has 128 characters: `\x00-\x7f`, a character class representing the lower half: `[\x00-\x3f]` can be represented by negating the upper half: `[^\x40-\x7f]`.

C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class belongs to the C4 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 6.1, though C3 may be able to transition to C1, C2 or C4 first and then to C5.

A pattern can belong to multiple representations. For example, `[a-f\d]` belongs to both C1 and C4. 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. This transformed version would only belong to the C1 node.

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]' ``.

We define understandability two ways. Assuming that common programming practices are more understandable than uncommon practices, we explore the frequencies of each representation from Figure 6.1 using thousands of regexes scraped from Python projects.

6.1.2 Implementation details

The goal of this study is to understand how frequently each of the regex representations appears in source code. Based on the results, we identify preferred representations using popularity in source code.

6.1.3 Artifacts

Artifacts were same as those described in chapter 3: Features. Adjust this to smoothly reflect the previous chapter

6.1.4 Metrics

6.1.5 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[1]. Next, we describe the process in detail:

6.1.5.1 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.

6.1.5.2 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

[1] https://github.com/softwarekitty/regex_readability_study

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

6.1.5.3 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).

6.2 Frequency analysis results

Table 6.1 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 6.1 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

Table 6.1: How frequently is each alternative expression style used?

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

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.

6.3 Discussion of representation frequency analysis

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 is more desirable based on community standards.

6.3.1 Context and common sense about these representations

6.3.2 Community support indicates a preference

6.3.3 Implications

6.3.4 Opportunities for future work

6.3.5 Threats to validity

CHAPTER 7. Comprehension of regex representations

7.1 Experiment design

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 6.1.

7.1.1 Metrics

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

Table 7.1: 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

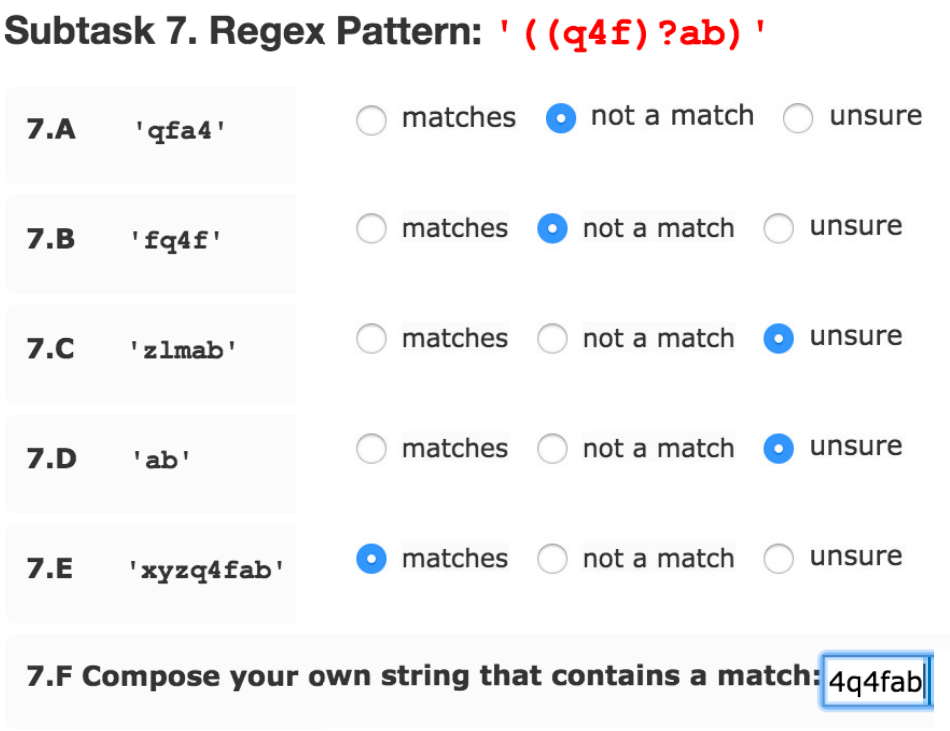

Figure 7.1: Example of one HIT Question

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 7.1.

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 7.1, 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. We refer to a response with all blank or unsure responses as an 'NA'. Out of 1800 questions, 1.8%(32) were NA's (never more than 4 out of 30 per pattern).

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.

7.1.2 Implementation

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.

7.1.2.1 Worker Qualification

Workers qualified to participate in the study 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.

7.1.2.2 Selecting pairwise comparisons

Using the patterns in the corpus as a guide, we created six metagroups containing three pairs of patterns focusing on:

- S1 vs S2
- the digit default character class vs C1
- the word default character class vs C1
- negated digits and words vs C3, whitespace vs C2
- additional vs kleene repetition
- wrapping vs escaping literal characters

and four metagroups containing two triplets of patterns focusing on

- octal vs hex vs literal
- D1 vs D2 vs D3
- C1 vs C2 vs C5
- octal vs literal and C2 vs C5

Each of these 10 metagroups contains 6 strings, resulting in a total of 60 regex patterns. These patterns are logically partitioned into 26 semantic equivalence groups (18 from pairs, 8 from triples).

Although we had 42 pairwise comparisons (18 from pairs, 24 from triples), 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 possible to attribute results purely to C2 and C5, or to 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.

Another example of a pairwise comparison from a pair used in this study is a group with regexes `` `([0-9]+)\.([0-9]+)' `` and `` `(\d+)\.(\d+)' ``, which is intended to evaluate the edge between C1 and C4. An example of pairwise compairisons from a triple is a semantic group with regexes `` `((q4f){0,1}ab)' ``, `` `((q4f)?ab)' ``, and `` `(q4fab|ab)' `` which is intended to explore the edges among D1, D2, and D3.

The end result is 36 pairwise comparisons across 14 edges from Figure 6.1.

7.1.2.3 Composing Tasks

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 7.1.

7.1.2.4 Worker outcomes

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.

1.

What is your gender?	n	%
Male	149	83%
Female	27	15%
Prefer not to say	4	2%

2. **What is your age?**
$\mu = 31$, $\sigma = 9.3$

3.

Education Level?	n	%
High School	5	3%
Some college, no degree	46	26%
Associates degree	14	8%
Bachelors degree	78	43%
Graduate degree	37	21%

4.

Familiarity with regexes?	n	%
Not familiar at all	5	3%
Somewhat not familiar	16	9%
Not sure	2	1%
Somewhat familiar	121	67%
Very familiar	36	20%

5. **How many regexes do you compose each year?**
$\mu = 67$, $\sigma = 173$

6. **How many regexes (not written by you) do you read each year?**
$\mu = 116$, $\sigma = 275$

Figure 7.2: Participant Profiles, $n = 180$ can remove this for space

Table 7.2: Averaged Info About Edges (sorted by lowest of either p-value)

Index	Nodes	Pairs	Match1	Match2	H_0^{match}	Compose1	Compose2	H_0^{comp}
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

7.2 Population characteristics

7.2.1 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. Participants read more regexes than they write with an average of 116 and a range from 0 to 2000. Figure 7.2 summarizes the self-reported participant characteristics from the qualification survey.

7.3 Matching and composition comprehension results

in study section present choices about pairwise vs random selection for nodes.

7.3.1 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 SectionN. This allowed us to compute and then average

26-30 values for each metric for each of the 60 regexes (fewer than 30 values were used if all the responses in a matching question were unsure or a combination of blanks and unsure).

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 and the direction of a refactoring for understandability. For example, one group had regexes, `RR*` and `R+`, which 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 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 than L2, from the perspective of both understandability metrics, suggesting a refactoring from L2 to L3.

This information is presented in summary in Table 7.2, with this specific example appearing in the E3 row. The *Index* column enumerates all the pairwise comparisons evaluated in this experiment, *Nodes* lists the two representations, *Pairs* shows how many comparisons were performed, *Match1* gives the overall matching score for the first representation listed and *Match2* gives the overall matching score for the second representation listed. H_0^{match} shows the results of using the Mann-Whitney test of means to compare the matching scores, testing the null hypothesis H_0: that $\mu_{match1} = \mu_{match2}$. The p-values from these tests are presented in this column. The last three columns list the average composition scores for the representations and the relevant p-value, also using the Mann-Whitney test of means.

Table 7.2 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.

Table 7.3: Average Unsure Responses Per Pattern By Node (fewer unsures on the left)

	¿=Q0(0.67)				¿=Q1(1.25)				¿=Q2(1.94)				¿=Q
Node	L3	D3	C2	C1	L2	S2	S1	C4	D1	C5	C3	D2	T1	T3
Number of Patterns	3	2	5	8	3	3	3	9	2	4	2	2	3	3
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

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 7.1). From a comprehension perspective, this indicates some level of confusion and is worth exploring.

For each pattern, we counted the number of responses containing at least one unsure, representing confusion. We then grouped the patterns into their representation nodes and computed an average of unsures per pattern. A higher number may indicate difficulty in comprehending a pattern from that node. Overall, the highest number of unsure responses came from T4 and T2, which present octal and hex representations of characters. The least number of unsure responses were in L3 and D3, which are both shown to be understandable by looking at E2 and E3 in Table 7.2.

These nodes and their average number of unsure responses are organized by quartile in Table 7.3. These results also corroborate the refactorings suggested by the understandability analysis for the LIT group (i.e., $\overrightarrow{T4T1}$), the DBB group (i.e., $\overrightarrow{D2D3}$), and the LWB group (i.e., $\overrightarrow{L2L3}$) because the more understandable node has the least unsures of its group. The findings for D3 and D2 are contradictory, however, as and further study is needed, 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 (i.e., 10 out of 30) was `` `xyz[\0133-\0140]' ``, so this may have been the least understandable pattern that we tested.

7.4 Discussion of comprehension results

7.4.1 Implications

7.4.2 Opportunities for future work

7.4.3 Threats to validity

Discuss the backslash explosions and readability issues!

CHAPTER 8. Topological sort of representations by frequency and comprehensibility

8.1 Design of topological sort

8.1.1 Conceptual Basis

8.1.2 Implementation details

8.2 Total ordering of representations

To determine the overall trends in the data, we created total orderings on the representation nodes in each equivalence class (Figure 6.1) with respect to the community standards and understandability metrics.

8.2.1 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 6.1 and the graphs for understandability are based on Table 7.2. The following sections describe the processes for building and and topologically sorting the graphs.

insert the rest of the graphs?

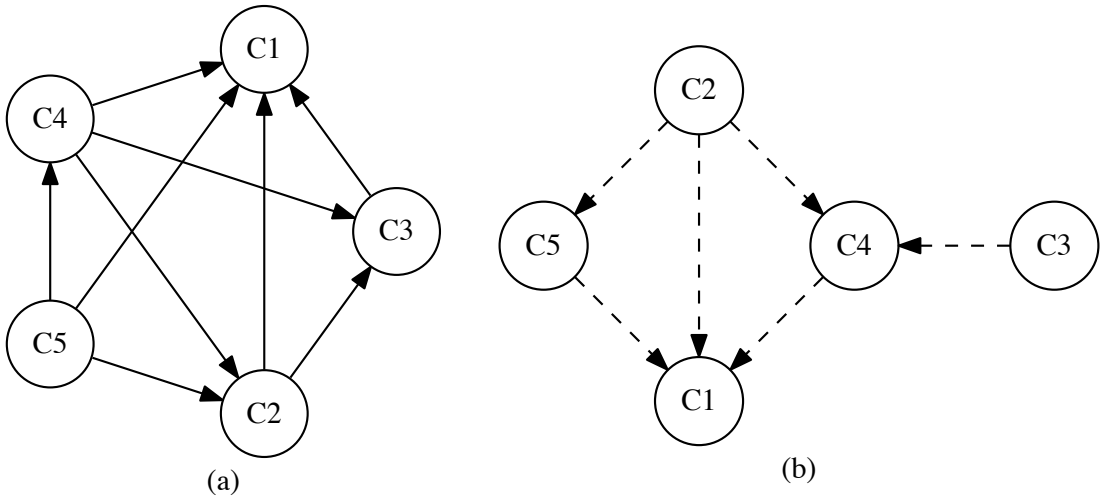

Figure 8.1: Trend graphs for the CCC equivalence graph: (a) represent the artifact analysis, (b) represent the understandability analysis.

8.2.1.1 Building the Graphs

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 6.1, we have created a graph, for example Figure 8.1a, that represents the frequency trends among the community artifacts. Note that with the CCC group, there is no edge between C3 and C5 because there is no straightforward refactoring between those representations, as discussed in Sectionɴ.

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 8.1b.

8.2.1.2 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.

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.

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 8.1b, 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

Table 8.1: 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

$n = C2$ 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 7.2, 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.

8.2.2 Results

After running the topological sort in Algorithm 1 with tiebreakers, we have a total ordering on nodes for each graph, shown in Table 8.1. For example, given the graphs in Figure 8.1a and Figure 8.1b, 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.

8.3 Discussion of ordering results

8.3.1 Implications

8.3.2 Opportunities for future work

8.3.3 Threats to validity

CHAPTER 9. DISCUSSION

9.1 Implications of the thesis as a whole

9.1.1 Review of implications already discussed

9.1.2 Implications considering all experiments together

9.2 Opportunities for future work studying regular expressions

9.2.1 Semantic search

9.2.2 Ephemeral regex

9.2.3 Comparing regex usage across communities

9.2.4 Evolution of patterns

9.2.5 Taxonomy of regex language varieties

9.2.5.1 Capturing Specific Content Near A Delimiter

The survey results from Section N indicate that capturing parts of strings is among the most frequent activities for which developers use regexes. From a feature perspective, the capture group (CG) is the most frequently used in terms of patterns (Table 3.1). This feature has two functions: 1) logical grouping as would be expected by parenthesis, and 2) retrieval of information in one logical grouping. As mentioned in Section N, capturing content was a primary goal evident in several cluster categories. The fourth-largest category is based entirely on capturing the content between brackets or parentheses (SectionN).

Many uses of CG also use the ANY and KLE features, eg. `(.*){(.*)}(.*)` and `\\s*([^: ]*)\\s*:(.*)`. This type of usage frequently revolves around an important delimiter character such as `:` or `\`.

This use case is well supported by existing tools for ASCII characters, but future tools should consider the centrality of this use case and its implications for non-English users of regex tools. For example, Unicode characters like 'U+060D' the Arabic Date Separator, or 'U+1806' the Mongolian Todo Soft Hyphen may be used to locate segments of text that a user would want to capture.

9.2.5.2 Counting Lines

Text files containing one unit of information per line are common in a wide variety of applications (for example .log and .csv files). Out of the 13,597 patterns in the corpus, 3,410 (25%) contained ANY followed by KLE (i.e., `` `.*' ``), often at the end of the pattern. One reasonable explanation for this tendency to put `` `.*' `` at the end of a pattern is that users want to disregard all matches after the first match on a single line in order to count how many distinct lines the match occurs on. Survey participants indicated an average frequency of "Counting lines that match a pattern" and "Counting substrings that match a pattern" at 3.2 or rarely/occasionally. It may be valuable for tool builders to include support for common activities such as line counting.

9.2.5.3 Refactoring Regexes

The survey showed that users want readability and find the lack of readable regexes to be a major pain point. This provides an opportunity to introduce refactoring transformations to enhance readability or comprehension. As one opportunity, certain character classes are logically equivalent and can be expressed differently, for example, `\d` ≡ `[0123456789]` ≡ `[0-9]`. While `\d` is more succinct, `[0-9]` may be easier to read, so a refactoring for *default to custom character classes* could be introduced. Human studies are needed to evaluate the readability and comprehension of various regex features in order to define and support appropriate regex refactorings.

Another avenue of refactoring could be for performance. Various implementations of regex libraries may perform more efficiently with some features than others. An evaluation of regex

feature implementation speeds would facilitate semantic transformations based on performance, similar to performance refactorings for LabVIEW Chambers and Scaffidi (2013, 2015).

Additionally, some developers may *find* specific content with a regex and then subsequently *capture* it with string parsing, which may be more error prone than using a capture group and indicates a missed opportunity to use the full extent of regex libraries. Future work will explore source code to identify the frequency of such occurrences and design refactorings to better utilize regex library features.

9.2.5.4 Migration Support for Developers

Within standard programming languages, regular expressions libraries are very common, yet there are subtle differences between language libraries in the supported features. For example, Java supports possessive quantifiers like `` `ab*+c' `` (here the '+' is modifying the '*' to make it possessive) whereas Python does not. Differences among programming language implementations was identified as a pain point for using regular expressions by 17% of the survey participants. This provides a future opportunity for tools that translate between regex utilizations in various languages.

9.2.5.5 Similarity Beyond String Matching

There are various ways to compute similarity between regexes, each with different trade-offs. While the similarity analysis we employ over-approximates similarity when compared to containment analysis, it may under-approximate similarity in another sense.

For example, two regexes that have dissimilar matching behavior could be very similar in purpose and in the eyes of the developer. For example, `commit:\[(\d+)\] - (.*)` and `push:\[(\d+)\] - (.*)` could both be used to capture the id and command from a versioning system, but match very different sets of strings. Future work would apply abstractions to the regex strings, such as removing or relaxing literals, prior to similarity analysis to capture and cluster such similarities.

From another perspective, our regex similarity measure, and even containment analysis, could treat behaviorally identical regexes as the same, when their usage in practice is completely

different. For example, in Table 4.1, the regexes `:+' and `(:+)' are behaviorally identical in that they match the same strings, except the latter uses a capture group. In practice, these may be used very differently, where the former may be used for validation and the latter for extraction. This usage difference could be observed by code analysis, and is left for future work.

9.2.5.6 Automated Regex Repair

Regular expression errors are common and have produced thousands of bug reports Spishak et al. (2012). This provides an opportunity to introduce automated repair techniques for regular expressions. Recent approaches to automated program repair rely on mutation operators to make small changes to source code and then re-run the test suite (e.g., Weimer et al. (2010); Le Goues et al. (2012)). In regular expressions, it is likely that the broken regex is close, semantically, to the desired regex. Syntax changes through mutation operators could lead to big changes in behavior, so we hypothesize that using the semantic clusters identified in Section N to identify potential repair candidates could efficiently and effectively converge on a repair candidate.

9.2.5.7 Developer Awareness of Best Practices

One category of clusters, *Content of Brackets and Parenthesis*, parses the contents of angle brackets, which may indicate developers are using regexes to parse HTML or XML. As the contents of angle brackets are usually unconstrained, regexes are a poor replacement for XML or HTML parsers. This may be a missed opportunity for the regex users to take advantage of more robust tools. More research is needed into how regex users discover best practices and how aware they are of how regexes should and should not be used.

9.2.5.8 Tool-Specific Regex Exploration

In some environments, such as command line or text editor, regexes are used extensively by the surveyed developers (Section N), but these regular expressions do not persist. Thus, using a repository analysis for feature usage only illustrates part of how regexes are used in

practice. Exploring how the feature usage differs between environments would help inform tool developers about how to best support regex usage in context, and is left for future work.

Based on our analyses of source code and our empirical study on the understandability of regex representations, we have identified preferred regex representations that may make regexes easier to understand and thus maintain. In this section, we describe the implications of these results.

9.2.6 Interpreting Results

In the CCC equivalence class, C1 (e.g., `[0-9a]`) is more commonly found in the patterns and projects. Representations C2 (e.g., `[0123456789a]`) and C3 (e.g., `[^\x00-/:-`b-\x7F]`) appear in similar percentages of patterns and projects but there is no significant difference in understandability considering two pairs of regexes tested as part of E13 (Table 7.2). However, a small preference is shown for C1 over C2 (E7), leading this to to be the winner of both the community support and understandability analyses. Regex length is probably important for understandability, though we did not test for this.

the longest regex in the corpus is X characters long...

In the DBB group, D3 (e.g., `pBs|pBBs|pBBBs`) merits further exploration because it is the most understandable but least common node in DBB group. This may be because explicitly listing the possibilities with an OR is easy to grasp, but if the number of items in the OR is too large, the understandability may go down. Further analysis is needed to determine the optimal thresholds for representing a regex as D3 compared to D1 (e.g., `pB{1,3}s`) or D2 (e.g., `pBB?B?s`).

Intuitively, it seems that D2 may be more common because 0,1 is just a more common use case than an arbitrary range like 4, 25.

In the SNG group, S1 is a compact representation (e.g., `S{3}`), but S2 was preferred (e.g., `SSS`). Similar to the DBB group, this may be do to the particular examples chosen in the analysis, as a large number of explicit repetitions may not be as preferred.

In the LWB group, L1 (e.g., `A{2,}`) is rare, appearing in $< 1\%$ of the patterns. Representations L2 (e.g., `AAA*`) and L3 (e.g., `AA+`) appear in similar numbers of patterns and projects,

but there is a significant difference in their understandability, favoring L3. it's clear that this is a rare use case, and also that L3 is the most common use case. Patterns using star are secondary, helper patterns because they will trivially match anything, so they are less common. But anyway...

S2 is over-weighted because of double-characters in regular words like foot. In the LIT group, T1 (e.g., `\a\$>`) is the typical way to list literals, but the reason to use hex (T2) or oct (T4) types is because some characters cannot be represented any other way, like invisible chars. One main result of our work is that T4 (e.g., `\007\036\062`) is less understandable than T2 (e.g., `\x07\x24\x3E`), so if invisible chars are required, hex is the more understandable representation. Regarding T3 (e.g., `\a[$]>`), initially we thought the square brackets would be more understandable than using an escape character, but we found the opposite. Given a choice between T1 and T3, the escape character was more understandable.

9.2.7 Opportunities For Future Work

There are several directions for future work related to regex study and refactoring.

Equivalence Class Models We looked at five equivalence classes, each with three to five nodes. Future work could consider richer models with more or different classes and nodes. For example, we have looked at all ranges as equivalent, all defaults as equivalent, and relied on many such generalizations. However, the range `[a-f]` is likely to be more understandable for most people than a range like `[:-`]`.

Additional equivalence groups to consider may include:

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

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

Case insensitive `(?i)[a-z]` ≡ `[A-Za-z]`

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

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

It might also be the case that there exist critical comprehension differences within a representation. For example, between C1 (e.g., `[0-9a]`) and C4 (e.g., `[\da]`), it could be the case that `[0-9]` is preferred to `[\d]`, but `[A-Za-z0-9_]` is not be preferred to `[\w]`). By creating

a more granular model of equivalence classes, and making sure to carefully evaluate alternative representations of the most frequently used specific patterns, additional useful refactorings could be identified.

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. 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 Migration Libraries We have identified opportunities 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 a promising direction of future work to ease the manual burden of this process, similar in spirit to prior work on class library migration Balaban et al. (2005).

Regex Refactoring Applications 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.

Regex Programming Standards 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.

Regex Refactoring for Performance The representation of regexes may have a strong impact on the runtime performance of a chosen regex engine. Prior work has sought to expedite the processing of regexes over large bodies of text Baeza-Yates and Gonnet (1996). Refactoring regexes for performance would complement those efforts. Further study is needed to determine which representations are most efficient, leading to a whole new area of study on regex refactoring for performance, a topic already explored for Depending on the efficiency of an organization's chosen regex engine, an organization may want to enforce standards for efficiency. , or for compatibility with a regex analysis tool like Z3, HAMPI, BRICS or REX.

9.2.8 Threats to Validity

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.

Participants evaluated regular expressions during tasks on MTurk, which may not be representative enough of the context in which programmers would encounter regexes in practive. Further study is needed to determine the impact of the experimentation context on the results.

Some regex representations from the equivalence classes were not involved in the understandability analysis and that may have biased the results against those nodes. Repetition of the analysis with more compete coverage of the edges in the equivalence classes is needed.

We treated unsure responses as omissions that did not count against the matching scores. Thus, if a participant answered two strings correctly and marked the other three strings as unsure, then this was 2/2 correct, not 2/5. This may have inflated the matching scores, however, less than 5% of the matching scores were impacted by such responses.

In our analyses, we measure understandability using matching and composition metrics. However, there may be other ways to approach regex understandability, such as deciding which regexes in a set are equivalent, finding the minimum modification to some text so that a given regex will match it. It may also be meaningful to provide some code that exists around a regex as context, since that would better represent a scenario in which programmers would encounter regexes in practice. Further study is needed to determine if the chosen metrics and experimentation context have resulted in a reasonable measure of understandability.

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 used in the evaluation were inspired by those 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.

The results of the understandability analysis may be closely tied to the particular regexes chosen for the experiment. For many of the representations, we had several comparisons. Still, replication with more regex patterns is needed.

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.

Our community analysis only focuses on the Python language, but as 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 and our results are likely to generalize. , 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.

Looks like M0, M1, M2, M3 and M9 are very dependent on the regex chosen, so regex-specific refactorings like:

0.1401 `&d([aeiou][aeiou])z'` `&d([aeiou]{2})z'` 0.075 `[\t\r\f\n ]'` `[\s]'` 0.1024 `[a-f]([0-9]+)[a-f]'` `[a-f](\d+)[a-f]'` 0.1271 `[\{][\$](\d+[.]\d)[}]'` `\\\{\\\$(\d+\.\d)\}'`

(from M0,M1,M2,M9 respectively) have okay P-values and may indicate regex-specific refactorings, but do not indicate an overall trend for that type of refactoring. Notice that M3 does not even have a strong p-value candidate, but this may be thrown off because of the very confusing regex chosen for CCC: 0.78 0.79 `xyz[_\[\]`\^\\]' xyz[\x5b-\x5f]'` which has a lot of escape characters, so that the hex group was easier to understand than the CCC.

Meanwhile M4,M5 and M7 have both ambiguous p-values and anova results. But this is still a finding: that no refactoring is needed between things like: `(q4fab|ab)' ((q4f){0,1}ab)' tri[abcdef]3' tri(a|b|c|d|e|f)3' &(\w+);' &([A-Za-z0-9_]+);'` (from M4,M5,M7 respectively) Although one refactoring from M5 might be of slight interest: 0.1196 FALSE `tri[a-f]3' tri(a|b|c|d|e|f)3'`

9.2.9 Reason for knowledge gap

clean and move this What explains the lack of rigor for regular expressions? I've been pondering this for quite a while now. This seems like it might be a case of being unable to see the forest because of the trees. Everyone in CS uses regex, and deals to some extent with other people using regex. It's never billed as top priority to optimize this, and it's so fundamental that the wikipedia pages explaining regex and Klene's theory are nearly circular with definitions of regular expressions relying on knowledge of what regular expressions are. The terminology is also very confusing to theory people, who probably believe that modern regex still represent DFA's and regular languages. I'd say there is an over-abundance of anecdotal evidence about how regex are used, so that people believe that it is a known topic. I also suspect a very real hacker/Unix ethos, where the odd ducks out there who really love regex also scorn formal evaluations and 'design by committee' in favor of just getting things to work, getting it good enough. It has flabbergasted me to look at every single documentation source for the dozens of different regex language 'flavors' and see that it is always just a thrown-together hodgepodge of examples and concepts. Most regex doc feels like it may be incomplete, saying stuff like 'it's mostly like PCRE'. Nobody and I mean nobody has a definitive feature list, outlining how the flavors differ and overlap. Like it's just okay to have all these dangling details for something that is pretty confusing already. Now you have two problems. It would be pretty fun to do a sort of gotcha-quiz of developers to see how many people would fall for the common myths and stuff.

CHAPTER 10. CONCLUSION

10.1 Summary of contributions

In an effort to find refactorings that improve the understandability of regexes, we created five equivalence class models and used these models to investigate the most common representations and most comprehensible representations per class. We found the most common representations per class by both number of patterns and number of projects to be C1, D2, T1 and S2 (L3 has the most patterns, L2 has the most projects). We also identified three strongly preferred transformations between representations (i.e., $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$) according to the results of our comprehension tests. We combined the results of these two investigations using a version of Kahn's topological sorting algorithm to produce a total ordering of representations within each model. The agreement between Community Standards and Understandability in this analysis validates our results and suggests that indeed one particular representation can be preferred over others in most cases. We can also recommend using hex to represent invisible characters in regexes instead of octal, and to escape special characters with slashes instead of wrapping them in brackets to avoid escaping them. Further research is needed into more granular models that treat common specific cases separately, and that address the effect of length on readability when transforming from one representation to another.

The contributions of this work are:

- A survey of 18 professional software developers about their experience with regular expressions,
- An empirical analysis of regex feature usage in nearly 14,000 regular expressions in 3,898 open-source Python projects, mapping of those features to those supported by common regex tools and survey results showing the impact of not supporting various features,

- An approach for measuring behavioral similarity of regular expressions and qualitative analysis of the most common behaviorally similar clusters, and
- An evidence-based discussion of opportunities for future work in supporting programmers who use regular expressions, including refactoring regexes, developing regex similarity analyses, and providing migration support between languages.

APPENDIX A. Patterns in Python projects from Github

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

top 100 clusters used

top 10 by feature group

13,579 patterns: the corpus(1510 per page=9 pages)

Supplemental material.

APPENDIX B. Developer Survey

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Survey Questions

Survey Responses

Survey Statistics

More stuff.

APPENDIX C. Mechanical Turk Study

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Qualifying Test

Template

MT Data input

MT All Results

MT Summary Statistics

More stuff.

APPENDIX D. Community Analysis

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Filter Criteria

Summary Statistics

More stuff.

BIBLIOGRAPHY

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 181–190. IEEE.

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2009). Extending sparql with regular expression patterns (for querying rdf). *Web Semant.*, 7(2):57–73.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and Mcminn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001.

Arslan, A. (2005). Multiple sequence alignment containing a sequence of regular expressions. In *Computational Intelligence in Bioinformatics and Computational Biology, 2005. CIBCB '05. Proceedings of the 2005 IEEE Symposium on*, pages 1–7.

Babbar, R. and Singh, N. (2010). Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data*, AND '10, pages 43–50, New York, NY, USA. ACM.

Baeza-Yates, R. A. and Gonnet, G. H. (1996). Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936.

Balaban, I., Tip, F., and Fuhrer, R. (2005). Refactoring support for class library migration. *SIGPLAN Not.*, 40(10):265–279.

Beck, F., Gulan, S., Biegel, B., Baltes, S., and Weiskopf, D. (2014). Regviz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 504–507, New York, NY, USA. ACM.

Begel, A., Khoo, Y. P., and Zimmermann, T. (2010). Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA. ACM.

Callaú, O., Robbes, R., Tanter, E., and Röthlisberger, D. (2011). How developers use the dynamic features of programming languages: The case of smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, New York, NY, USA. ACM.

Callaú, O., Robbes, R., Tanter, E., and Röthlisberger, D. (2013). How (and why) developers use the dynamic features of programming languages: The case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194.

Chambers, C. and Scaffidi, C. (2013). Smell-driven performance analysis for end-user programmers. In *Proc. of VLH/CC '13*, pages 159–166.

Chambers, C. and Scaffidi, C. (2015). Impact and utility of smell-driven performance tuning for end-user programmers. *Journal of Visual Languages & Computing*, 28:176–194. to appear.

Chapman, C. and Stolee, K. T. (2016). Exploring regular expression usage and context in python. under review.

Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014). An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91, New York, NY, USA. ACM.

Dattero, R. and Galup, S. D. (2004). Programming languages and gender. *Commun. ACM*, 47(1):99–102.

Du Bois, B., Demeyer, S., Verelst, J., Mens, T., and Temmerman, M. (2006). Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355.

Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. (2014). Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA. ACM.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Galler, S. J. and Aichernig, B. K. (2014). Survey on test data generation tools. *Int. J. Softw. Tools Technol. Transf.*, 16(6):727–751.

Ghosh, I., Shafiei, N., Li, G., and Chiang, W.-F. (2013). Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA. IEEE Press.

Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., and Ghezzi, C. (2010). An empirical investigation into a large-scale java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA. ACM.

Griswold, W. G. and Notkin, D. (1993). Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269.

Hermans, F., Pinzger, M., and van Deursen, A. (2012). Detecting code smells in spreadsheet formulas. In *Proc. of ICSM '12*, pages 409–418.

Hermans, F., Pinzger, M., and van Deursen, A. (2014). Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, pages 1–27.

Kiezun, A., Ganesh, V., Artzi, S., Guo, P. J., Hooimeijer, P., and Ernst, M. D. (2013). Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72.

Lee, J., Pham, M.-D., Lee, J., Han, W.-S., Cho, H., Yu, H., and Lee, J.-H. (2010). Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth International Workshop on Data and Text Mining in Biomedical Informatics*, DTMBIO '10, pages 23–30, New York, NY, USA. ACM.

Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., and Jagadish, H. V. (2008). Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA. Association for Computational Linguistics.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA. ACM.

Livshits, B., Whaley, J., and Lam, M. S. (2005). Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg. Springer-Verlag.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Trans. Soft. Eng.*, 30(2):126–139.

Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 1–18, New York, NY, USA. ACM.

Møller, A. (2010). dk.brics.automaton – finite-state automata and regular expressions for Java. `http://www.brics.dk/automaton/`.

network (2015). The Bro Network Security Monitor. https://www.bro.org/.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA. UMI Order No. GAX93-05645.

Parnin, C., Bird, C., and Murphy-Hill, E. (2013). Adoption and use of java generics. *Empirical Softw. Engg.*, 18(6):1047–1089.

re2 (2015). RE2. https://github.com/google/re2.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12.

Spishak, E., Dietl, W., and Ernst, M. D. (2012). A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA. ACM.

Stolee, K. T. and Elbaum, S. (2011). Refactoring pipe-like mashups for end-user programmers. In *International Conference on Software Engineering*.

Stolee, K. T. and Elbaum, S. (2013). Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Softw. Eng.*, 39(12):1654–1679.

Tillmann, N., de Halleux, J., and Xie, T. (2014). Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA. ACM.

Trinh, M.-T., Chu, D.-H., and Jaffar, J. (2014). S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA. ACM.

Veanes, M., Halleux, P. d., and Tillmann, N. (2010). Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA. IEEE Computer Society.

Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight*, 53(5):109–116.

Yeole, A. S. and Meshram, B. B. (2011). Analysis of different technique for detection of sql injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 963–966, New York, NY, USA. ACM.