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 vii

LIST OF FIGURES viii

ACKNOWLEDGEMENTS ix

ABSTRACT i

CHAPTER 1. OVERVIEW 1

1.1 About this thesis 1

1.2 Regular expression basics 2

1.2.1 Language nomenclature 2

1.2.2 Matching strings defined 2

1.2.3 Patterns are not regexes 3

1.2.4 Most patterns compile to a functionally identical regex in most languages 3

1.3 Description of studied features 4

1.3.1 Elements 5

1.3.2 Options 9

1.3.3 Operators 9

1.3.4 Positions 11

1.4 Milestones in regular expression history 13

1.4.1 Kleene's theory of regular events 13

1.4.2 First regex compiler 13

1.4.3 Early regular expressions in Unix 14

1.4.4 Maturity of standards 14

CHAPTER 2. RELATED WORK 15

2.1 Applications of regex 15

2.1.1 Programming languages that support regex 15

2.2 Analyzing and testing regex 18

2.3 Composing Assistants 18

2.4 Special Applications for regex 19

2.5 Formalisms and research addressing regex 19

2.6 Gap in fundamental research into regex use in practice 19

2.7 Questions explored in this thesis and their motivations 20

2.7.1 RQ1: How are regex used in practice, especially what features are most commonly used? 20

2.7.2 RQ2: What behavioral categories can be observed in regex? 20

2.7.3 RQ3: What preferences, behaviors and opinions do professional developers have about using regex? 20

2.7.4 RQ4: Within five equivalence classes, what representations are most frequently observed? 21

2.7.5 RQ5: What representations are more comprehensible? 21

2.7.6 RQ6: For each equivalence class, which representation is preferred according to frequency and comprehensibility? 21

2.8 Surveys of regex research 21

2.9 Mining 21

2.10 Refactoring and smells 22

CHAPTER 3. Feature Analysis 24

3.1 Overview of feature frequency experiment 24

3.2 Utilizations of the re module 25

3.2.1 Utilization defined 25

3.2.2 Omission of calls to compiled objects 26

3.3 Github mining implementation 26

3.3.1 Objects used in design 26

3.3.2 Mining Algorithm . . . 27
3.3.3 Database schema . . . 30
3.3.4 Challenges in implementation . . . 31
3.4 Building the corpus of regexes . . . 32
3.4.1 Selecting projects to mine for utilizations . . . 32
3.4.2 Saturation of artifacts with regexes . . . 33
3.4.3 Flags and functions . . . 34
3.4.4 Selecting a body of patterns from a set of utilizations . . . 35
3.4.5 Parsing Python Regular Expression patterns using a PCRE parser . . . 35
3.5 Analyzing the corpus of regexes . . . 36
3.5.1 Parsing feature tokens . . . 36
3.5.2 Feature usage within the corpus . . . 38
3.6 Feature Support . . . 39
3.6.1 Caveats to consider when comparing feature sets . . . 39
3.6.2 Choosing languages to compare feature support . . . 40
3.6.3 Ranked feature support . . . 40
3.6.4 Alien feature support . . . 41
3.6.5 Alien feature descriptions . . . 41
3.6.6 Feature support in regex analysis tools . . . 43
3.7 Discussion of utilization and feature analysis results . . . 44
3.7.1 Implications . . . 44
3.7.2 Opportunities for future work . . . 44
3.7.3 Threats to validity . . . 45

CHAPTER 4. Equivalent representations of regex and their frequencies in the corpus . . . 49

4.1 Defining five equivalence classes . . . 49
4.1.1 Conceptual Overview . . . 49
4.1.2 CCC Group . . . 51
4.1.3 DBB Group . . . 52

4.1.4 LIT Group . . . 53
4.1.5 LWB Group . . . 54
4.1.6 SNG Group . . . 54
4.1.7 Example regex . . . 55
4.2 Counting representations in nodes . . . 55
4.2.1 Overview of the node counting process . . . 55
4.2.2 Artifact details . . . 56
4.2.3 Implementation details of determining node membership . . . 57
4.3 Node counting results . . . 59
4.4 Discussion . . . 60
4.4.1 Preferences indicated by community support . . . 60
4.4.2 Opportunities for future work . . . 65
4.4.3 Threats to validity . . . 66
CHAPTER 5. Comprehension of regex representations . . . 67
5.1 Experiment design . . . 67
5.1.1 Metrics . . . 67
5.1.2 Implementation . . . 69
5.2 Population characteristics . . . 72
5.2.1 Participants . . . 72
5.3 Matching and composition comprehension results . . . 74
5.3.1 Analysis . . . 74
5.4 Discussion of comprehension results . . . 77
5.4.1 Implications . . . 77
5.4.2 Opportunities for future work . . . 78
5.4.3 Threats to validity . . . 78

LIST OF TABLES

1.1 Codes, descriptions and examples of select Python Regular Expression features 4

2.1 Regex-based feature breakdown for 10 popular code editing tools . . . 16

2.2 Regex-based feature descriptions for 7 popular command line tools . . 16

2.3 Regex-based feature descriptions for 5 popular sql engines 16

3.1 How saturated are projects with utilizations? 33

3.2 Frequency of feature appearance in Projects, Files and Patterns, with number of tokens observed and the maximum number of tokens observed in a single pattern. 37

3.3 What regular expression languages support features studied in this thesis? 46

3.4 What other features are supported in various languages? 47

3.5 What features are supported by regular expression analysis tools? . . . 48

4.1 How frequently is each alternative expression style used? 60

5.1 Matching metric example 67

5.2 Averaged Info About Edges (sorted by lowest of either p-value) 74

5.3 Equivalent regexes with a significant difference in readability 75

5.4 Average Unsure Responses Per Pattern By Node (fewer unsures are lower) 76

LIST OF FIGURES

3.1 Example of one regex utilization 25

3.2 Which behavioral flags are used? 34

3.3 How often are `re` functions used? 34

3.4 Two patterns parsed into feature vectors 38

4.1 Equivalence classes with various representations of semantically equivalent representations within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal 50

5.1 Example of one HIT Question 68

5.2 Participant Profiles, $n = 180$ 73

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.

1.2.4 Most patterns compile to a functionally identical regex in most languages

Examples have been shown of patterns that have alternate meanings depending on the regular expression language, and patterns that can be compiled by one engine, but not another. However, it is typical for a pattern using common features to compile to a regex with identical behavior in different regular expression languages. The extent of feature variation among

languages is a difficult subject to approach, and so it is difficult to quantify how often this occurs. It is likely that many programmers not well versed in the nuances of regular expressions believe that all patterns can be used with identical effects in all languages. An attempt to document what features are supported by what languages is provided in Section **??**.

1.3 Description of studied features

Table 1.1 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`

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:

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. Every character can also be expressed in hex or octal form. For example, a regex expressing the newline character `\n` is equivalent to the same character expressed in hex: `\x0A` and octal: `\012`. In addition to the characters mentioned, the singlequote `` `'' `` and doublequote `` `"' `` often must be escaped, depending on the quotation style used in source code. This work will not address this issue in further detail.

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"`.

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

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.

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 `(?s)` for single-line mode making ANY match all characters, `(?m)` for multiline mode, making STR and END match the beginning and ending of every line, `(?l)` may change the meaning of default character classes if a locale has been set, `(?x)` ignores whitespace between *tokens* 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"`.

The syntax of this feature may cause much confusion when porting to another language like Java, Perl, JavaScript, etc. where the lowercase z: `\z` has this meaning, but the uppercase Z: `\Z` *would* match `"R\n"` - it matches the end of string or before the last newline.

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.

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

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 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. The std::regex library[3] implements engines for ECMAScript Regular Expressions (the default - same is used by JavaScript), AWK Regular Expressions, POSIX BRE or POSIX ERE. The following libraries are alternatives to std::regex libary for C++: 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 **?**) 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

[3] http://en.cppreference.com/w/cpp/regex/syntax_option_type

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 **?**) or to allow visual debugging **?**). Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text **??**).

Research tools like Hampi **?**), and Rex **?**), and commercial tools like brics **?**) 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 **??**). 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.[4] Others will automatically generate strings that are matched by the regular expessions **?**). Other tools will automatically generate regexes when given a list of strings to match **??**). The commonality of such tools provides evidence that people need help with regex composition and understandability.

2.3 Composing Assistants

VerbalExpressions[5].

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

2.4 Special Applications for regex

Some data mining frameworks use regular expressions as queries (e.g., **?**)). Efforts have also been made to expedite the processing of regular expressions on large bodies of text **?**).

Regarding applications, regular expressions have been used for test case generation **????**), and as specifications for string constraint solvers **??**). Regexes are also employed in MySQL injection prevention **?**) and network intrusion detection **?**), or in more diverse applications like DNA sequencing alignment **?**) or querying RDF data **??**).

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 **?**). To aid in regex creation and understanding, tools have been developed to support more robust creation **?**) or to allow visual debugging **?**). Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text **??**).

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 **?**), JavaScript **?**), and Java **????**), and more specifically, Java generics **?**) and Java reflection **?**). Our prior work (**?**), under review) was the first to mine and evaluate regular expression usages from existing software repositories. The intention of the prior work **?**) was to explore regex language features usage and surveyed developers about regex usage. In this work,

we define potential refactorings and use the mined corpus to find support for the presence of various regex representations in the wild. Beyond that, we measure regex understandability and suggest canonical representations for regexes to enhance conformance to community standards and understandability.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns **?**) and bug characterizations **?**). Exploring language feature usage by mining source code has been studied extensively for Smalltalk **??**), JavaScript **?**), and Java **????**), and more specifically, Java generics **?**) and Java reflection **?**). 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 **?**), but have not been the focus of the mining activities. Surveys have been used to measure adoption of various programming languages **??**), and been combined with repository analysis **?**), but have not focused on regexes.

2.10 Refactoring and smells

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

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

Code smells in object-oriented languages were introduced by Fowler **?**). Researchers have studied the impact of code smells on program comprehension **??**), finding that the more smells in the code, the harder the comprehension. This is similar to our work, except we aim to identify which regex representations can be considered smelly. Code smells have been extended to other language paradigms including end-user programming languages **????**). The code smells identified in this work are representations that are not common or not well understood

by developers. This concept of using community standards to define smells has been used in other refactoring literature for end-user programmers **??**).

CHAPTER 3. Feature Analysis

3.1 Overview of feature frequency experiment

The primary goal of this experiment was to determine the frequency with which Python Regular Expression features are used in the wild. In order to obtain data about feature usage frequency, a large number of patterns used to create regexes were required. One obvious place to obtain these patterns was by looking at source code that calls the `re` module. One call to this module found in source code (not running live) will be referred to as a *utilization*. Utilizations are explained in further detail in Section 3.2

With these needs in mind, a tool was implemented that does the following:

- finds projects containing Python on Github
- clones the repositories containing these projects
- builds the AST of source code using files from these projects
- populates a database with information about utilizations found

Implementation details of this tool, and some of the challenges faced are discussed in Section 3.3. Once the data about utilizations had been collected, some questions about the utilizations themselves were explored. This exploration can be read about in Section 3.2.

The patterns obtained from the utilizations were parsed using a PCRE parser to create Table **??**. This table summarizes the findings of this experiment, that is, for each feature described in Section 1.3, this table shows the number of patterns containing that feature, and the number of projects using that feature in some pattern (as well as other data). These findings are presented in Section **??**.

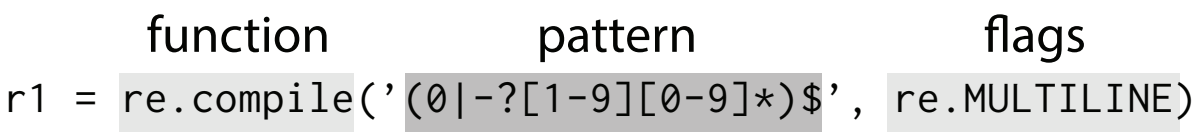

Figure 3.1 Example of one regex utilization

With the knowledge of how frequently each feature is used, the sets of features supported by various regular expression analysis tools becomes more interesting. A moderate survey exploring supported features can be found in Section **??**. Finally a discussion of the impact of this study, opportunities for future work and threats to validity can be found in Section 3.7.

3.2 Utilizations of the re module

Utilization: A *utilization* occurs whenever a regex is used in source code. We detect utilizations by statically analyzing source code and recording calls to the `re` module in Python.

3.2.1 Utilization defined

Within a Python source code file, a utilization of the `re` module is composed of a function, a pattern, and 0 or more flags. Figure 3.1 presents an example of one utilization, with key components labeled. The function call is `re.compile`, `"(0|-?[1-9][0-9]*)$"` is the pattern, and `re.MULTILINE` is an (optional) flag. When executed, this utilization will compile a regex into the variable `r1` from the pattern `"(0|-?[1-9][0-9]*)$"`. The resulting regex `(0|-?[1-9][0-9]*)$` is composed of two regex fragments: `0` and `-?[1-9][0-9]*` operated on by the OR `|`, and contained in a CG `( )` so that the following the END feature (`$`) applies regardless of which fragment is matched. Because of the `re.MULTILINE` flag used, the END specifies a position at the end of every line (instead of only the end of the last line).

The regex fragment `0` matches `"0"`, and the fragment on the right of the OR, `-?[1-9][0-9]*`, matches all positive or negative integers (not starting with 0) like `"123"`, `"9"`, `"-10000"` or `"-8"`. When combined the full regex `(0|-?[1-9][0-9]*)$` matches all positive and negative integers at the end of lines. For example the multi-line string: `"line 1: xyz 85\nline2: -2\nlast line\n"` will match at the end of the first two lines. Expressing zero or one dash characters using the

regex fragment `-?` is useful so that the sign of the integer will be part of the capture, (e.g., from `"A: -9"\n`, `"-9"` is captured, not just `"9"`).

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 3.1. As described in Section 1.2.1, a pattern is an ordered series of regular expression language feature tokens which can be compiled by an engine into a regex. A regex compiled from the pattern in Figure 3.1 .

Note that because the vast majority of regular expression 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 as mentioned in Section **??**, whereas a utilization is not universal in the same way (i.e., it is very unlikely to compile in other languages because of variations in programming language syntax and the names of functions).

3.2.2 Omission of calls to compiled objects

Every utilization recorded using the technique described in Section **??** is an invocation directly using the `re` library, like `re.compile(...)` or `re.search(...)`. However, this technique is not able to record calls on compiled objects. For example the regex described in Section 3.2.1 is stored in the variable `r1`. The regex in this variable can be used to call `re` module functions but will not use the re library directly. For example the code `r1.search("-45")` would not be recorded by the technique used in this work. However, since our primary focus is on patterns and the features of their compiled regexes, and these patterns must be compiled before becoming regexes, this omission only impacts the interpretation of Figure 3.3, which describes which function calls were observed. Notice that *every compilation of a pattern to a regex* is captured by the technique used in this study.

3.3 Github mining implementation

The Github mining tool, named `tour_de_source` was written in an object-oriented style by a programmer with relatively little experience in Python.

3.3.1 Objects used in design

The mining process is conducted using the following four objects:

Scanner provides the `scanDirectory()` function, which scans a directory, recording utilizations. This object also tracks the total number of projects scanned and the frequency of the number of files scanned per project.

Rewinder handle for a particular repository. The `getUniqueSourceID()` and `getSourceJSON()` functions provide metadata about the repository, and the `rewind()` function resets a repository to an earlier state in its history.

Sourcer handle for a source of projects. The `next()` function gets a rewinder for the next Python project, and the `isExhausted()` function returns true if there are no more projects. The sourcer also tracks the total number of projects checked for Python source code.

Tourist provides the `tour()` function which controls the mining process.

3.3.2 Mining Algorithm

The algorithm used for mining is quite straightforward, but the tour() 1 and scanDirectory() 2 functions are described here for reference (with logging, profiling and exception handling functionality removed, and some changes for readability).

Algorithm 1 The tour() function

```
1:  while not sourcer.isExhausted() do
2:      rewinder = sourcer.next()
3:      filePathSet = []
4:      uniqueSourceID = rewinder.getUniqueSourceID()
5:      sourceJSON = rewinder.getSourceJSON()
6:      while ( dorewinder.rewind())
7:          scanner.scanDirectory(uniqueSourceID, sourceJSON, filePathSet)
8:      end while
9:      nFiles = len(filePathSet)
10:     scanner.incrementNFilesFrequencies(nFiles)
11:     scanner.incrementNProjectsScanned()
12: end while
```

Iterating through projects The `tour()` function 1 simply iterates through available sources, using the `isExhausted()` function on Line 1 to check that another source is available, and then using the `next()` function on Line 2 to get the a rewinder object that handles the

current repository. Internally, the `next()` function pages through all repositories on Github using the `https://api.github.com/repositories?since=<lastRepoID>` endpoint to get a page describing 100 repositories. Each project description contains a url endpoint containing a description of the languages that the project contains. This url is visited and if it indicates that the project contains Python, then the a rewinder is created for that project. Note that the language url is automatically maintained by Github - developers do not have to go through any steps to indicate that a project contains Python, aside from committing a file written in Python.

Creating a rewinder The name, clone url and other metadata for a repository containing Python is collected using the Github API, and then cloned into a new directory named using the repoID provided by Github to ensure uniqueness. A list of commit logs is parsed, gathering the date and SHA of all commits. If a project has 20 or fewer commits, all of them are added to a stack and the rewinder is complete. Otherwise the most recent commit is added to a stack, and unit spacing is computed by dividing the number of remaining commits by 19, and 19 more evenly-spaced commits are added to the stack.

Rewinding through commit history On Line 6 the rewinder attempts to rewind the repository through a history of commits. Internally the rewinder uses the `git` Python module to perform `git reset --hard <SHA>`, and will return true unless it has reached the end of its list of 20 or fewer commit SHAs.

Rationale for using 20 commits The idea of using 20 commit points is that the patterns within utilizations may change over time, but with some experimentation this was determined to not happen very often. The number of commits to use was selected by trial and error and attempts to balance the time and memory used to build the AST with the more expensive operation of finding and cloning an entire project for the first time.

Scanning the project at one point in history On Line 7 the scanner is called with metadata about the current project commit and an empty list for tracking file paths. This

Algorithm 2 The scanDirectory() function

```
1: uniqueSourceID, sourceJSON, filePathSet passed as arguments
2: shaSet = []
3: citationSet = []
4: pythonAbsPaths = get absolute paths of files in repo directory ending in '.py'
5: for f doileAbsPath in pythonAbsPaths
6:     fileHash = util.getHash(fileAbsPath)
7:     if fileHash not in shaSet then
8:         shaSet.append(fileHash)
9:         fileRelPath = get relative file path from fileAbsPath
10:        if fileRelPath not in filePathSet then
11:            filePathSet.append(fileRelPath)
12:        end if
13:        root = astroid.ast_from_file(relFilePath)
14:        metadata = struct(uniqueSourceID, sourceJSON, fileHash, fileRelPath)
15:        extractRegexR(root, metadata, citationSet)
16:    end if
17: end for
```

scanning function is described in Algorithm 2. In addition to the metadata and file path list passed to scanner, an empty list of sha strings and another empty list of citations are created on Line 2 and Line 3, respectively. These lists are used to avoid re-scanning duplicate files as well as tracking duplicate utilizations and total number of files scanned. The lists are sets in practice, because no element is added without first checking if the list contains it.

On Line 4, a list of absolute paths of Python files in the repository is created. Iteration over this list begins on Line 5. For each of these files a SHA_224 of the file is computed (on Line 6) using Python's `hashlib` module like `hashlib.sha224(fileContents)` (and converted to a base 36 string for readability). It is unlikely that two files with different content will map to the same SHA_224, and impossible for the same content to map to two different SHA_224 strings. If the fileHash is not already in the shaSet, then it is assumed this exact file content has not been scanned yet. Unique relative file paths are added to the filePathSet for tracking on Line 11. The `astroid` module is used on Line **??** to build an AST of the source code contained in the current Python file, and the root of the tree is stored in a variable. This root, the metadata about the current project commit, and the citationSet are passed to the `extractRegexR` function on Line 15.

Extracting utilizations from an AST The extractRegexR function is tightly bound to the internal details of the `astroid` module, which is fairly complex and verbose, so no Algorithm is shown. Little documentation exists on how to use `astroid` to extract utilizations, so the technique used was developed by trial and error on a test project known to contain every type of utilization of interest. The details of each utilization was internally treated as a 4-tuple called a 'citation', containing:

1. The relative file path.
2. The name of the function of the `re` module called.
3. The pattern in the utilization.
4. The flags as an integer formed using a bitmask.

If the citationSet already contained a duplicate 4-tuple, the new citation was not added to the citationSet. Otherwise the citation represented a unique utilization, and so was recorded in the database along with relevant metadata. Multiple runs on multiple machines were completed to collect the utilizations used to build the corpus. Each run produced its own database file, and so after enough data had been collected, the data from all runs was merged into a single database.

3.3.3 Database schema

Early implementations of tour_de_source stored project metadata in a separate table. This led to awkward and verbose queries, and so the final version used only two tables: `RegexCitationMerged` and `FilesPerProjectMerged`. The `FilesPerProjectMerged` table has two columns of integers: `nFiles` and `frequency` - these were used to generate statistics about how many Python files the scanned projects contained. The columns of the `RegexCitationMerged` table are described below:

uniqueSourceID An ID generated by tour_de_source (sequentially) for each source.

repoID The ID of the repository on Github.

sourceJSON something here

fileHash The SHA_224 hash of the file containing the utilization.

filePath The path of the file containing the utilization (relative to the repository root).

pattern The string compiled into a regex in the utilization.

flags An integer representing the 6 flags as described in elsewhere?

regexFunction The name of the function called in the utilization.

3.3.4 Challenges in implementation

Python garbage collector ignores integers It was a surprise to find out that the memory used by tour_de_source only grew as mining went on. Every time that astroid built a new AST, memory consumed would climb by many megabytes, with jumps as large as 350 megabytes observed. The machines running tour_de_source only had 16 gigabytes of memory, and so they could only mine utilizations from a few hundred projects before failing. Every effort was made to profile the system and find a memory leak, without positive results. The only viable explanation found is that an AST can have a very large number of nodes, each identified by a unique integer, and none of the memory used to store these integers is reclaimed after the maps go out of scope.

Rationale behind building the AST The tool used to mine utilizations from Python project was written in Python to take advantage of the astroid reference library, which is a Python AST parser that is actively being maintained in order to support Pylint reference. The decision to use an AST parser instead of, say, trying to extract utilizations using a regex, was made due to the difficulty of writing a regex that cannot be fooled into capturing the wrong content. A simple example is some source code like `re.compile("d̈")`, which a naive regex like `re.compile\("([^)])"\)` would capture the string `"\"` instead of the actual content `"\")"`.

Erasing cloned files Every effort was made to erase a repository once scanning was complete, but for whatever reason, certain files could not be erased automatically. Some files seemed to have read-only flags set, and occasionally the file system lock for that file had been obtained by another process (probably git) but never released. These errors caused

unexpectedly serious problems - when a repository failed to erase completely, a new repository was not cloned, meaning that no '.git' folder was present in the target directory. As a result, the '.git' folder of the tour_de_source project itself was referenced by calls to git, causing the source code of the mining tool to be rewound by the mining tool! The solution to this problem was to allow the files to remain and erase them using the command line later.

Github API rate limit and network latency The mining program was able to check about one repository ID per second, which was slowed by network latencies, or, once 5000 API calls had been made in one hour, was throttled by Github. The apparent solution was to create multiple accounts, each providing 5000 API calls per hour. After contacting Github to request help with this issue, they indicated that they do not want users to create multiple accounts for mining projects because it can put a strain on their servers, slowing the service for regular users. An alternative strategy was proposed by Github of using a database to find Python projects without using the API. However, at this point in the project, enough data had been acquired to begin analysis and a determination was made to stop development of this mining program and focus on analysis. Future mining efforts are encouraged to obtain repository information from this database instead of crawling through all projects using the Github API, like tour_de_source did.

3.4 Building the corpus of regexes

The goal of this experiment was to collect regexes from a variety of projects to represent the breadth of how developers use the language features.

3.4.1 Selecting projects to mine for utilizations

Using the GitHub API, 3,898 projects containing Python code were mined for utilizations as described in Section 3.3. This section describes how these projects were selected.

Every time a new repository is created on Github, a new unique identifier (strictly greater than existing identifiers) is generated and assigned to that repository. This work refers to these identifiers using the shorthand: *repo ID*. At the time the mining for utilizations used in this

Table 3.1 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

study was performed, the largest repo ID was between 32 million and 33 million. Dividing these repo IDs into four groups each of size $2^23 = 8,388,608$ (with the fourth group being a little larger than that), the second group, which spans the range 8,388,608 - 16,777,215 was split into 32 sections so that starting indices were 262,144 repo IDs apart. The original intention was to mine the entire second 1/4 of the first 32 million repo IDs, but due to the challenges described in Section 3.3.4, only the first 100 or so projects from each of the 32 starting points was mined. Instead of spending the majority of available time on perfecting a mining technique, the determination was made to analyze the data that had already been gathered.

3.4.2 Saturation of artifacts with regexes

Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one utilization. Within a project, a duplicate utilization was marked when two versions of the same file have the same function, pattern and flags. In total, 53,894 non-duplicate utilizations were observed. To illustrate how saturated projects are with regexes, measurements are made for the number of utilizations per project, number of files scanned per project, number of files containing utilizations, and number of utilizations per file, as shown in Table 3.1.

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[1] 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 3.1,

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

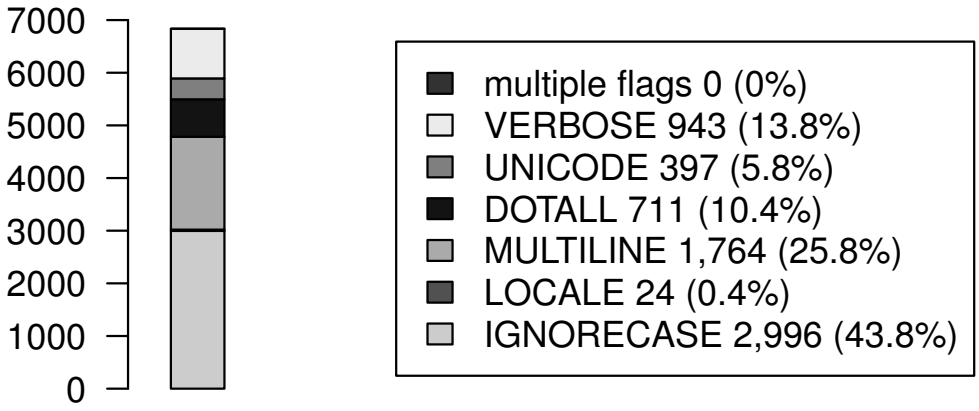

Figure 3.2 Which behavioral flags are used?

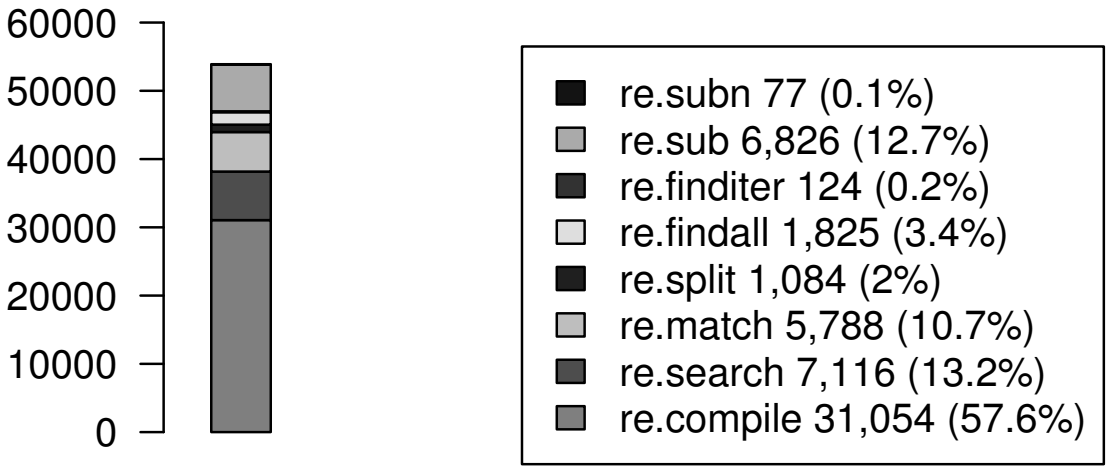

Figure 3.3 How often are `re` functions used?

it can also be seen that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations.

3.4.3 Flags and functions

As shown in figure 3.2, of all behavioral flags used, ignorecase (43.8%) and multiline (25.8%) were the most frequently used. It is also worth noting that although multiple flags can be combined using a bitwise or, this was never observed. When considering flag use, non-behavioral flags (default and debug) were excluded, which are present in 87.3% of all *utilizations*.

As seen in Figure 3.3 The 'compile' function encompasses 57.6% of all utilizations. Regexes may be compiled in an attempt to improve performance (only compile once) or to abstract the regex from the rest of the code. Compiled regexes are often observed at the top of a file, listed along with other highly-scoped variables maintained separately from blocks of code. Using the other `re` module functions in-line may be less preferred by developers because of the 'magic strings' which could be refactored to a variable.

3.4.4 Selecting a body of patterns from a set of utilizations

To guarantee that the behavior of regexes used for analysis depended only on the pattern extracted from a utilization, the 12.7% of utilizations using flags were excluded from further analysis. 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). All distinct patterns from the remaining 80.8% (43,525) of utilizations were pre-processed by removing Python quotes (`'\\W'` becomes `\\W`), and unescaping escaped characters (`\\W` becomes `\W`). After these filtering steps, 13,711 distinct patterns remained.

3.4.5 Parsing Python Regular Expression patterns using a PCRE parser

The collection of distinct patterns formed by this process was parsed into tokens using an ANTLR-based, open source PCRE parser[2]. A comparison of the features supported by this parser (Perl features) and Python is provided in Table 3.3, and indicates that all but the ENDZ feature have identical syntax and meaning. Fortunately, the syntax of the ENDZ feature (e.g., `R\Z`) matches the syntax of the LNLZ feature (e.g., `R\Z`) so that in practice, the parser used can correctly identify all studied features. To clarify the difference, if a newline is the last character in a string, ENDZ will match after that newline, and LNLZ will match before that newline.

This parser was unable to support 0.5% (73) of the patterns due to unsupported Unicode characters. Another 0.1% (17) of the patterns used PCRE features not valid in Python (see Section match with section for more information on these features). Two additional patterns used

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

the commenting feature which is valid in Python but encountered so rarely that it is not included in the analysis of features.

Details about the patterns excluded due to alien features used are provided here:

IFC (If conditionals) six patterns like `"^(\()?([^()]+)(?(1)\))$"`

NCND (Named conditions) five patterns like `"(?P<g2>b)?((?(g2)c|d))"`

IFEC (If-else conditionals) three patterns like `"^(?:(a)|c)((?(1)b|d))$"`

ECOM (Comments) two patterns like `"(?# Break or beginning)"`

LHX (Long hex) two patterns like `"\uFF0E"`

PXCC (Posix character classes) one pattern containing the fragment `"([[:alpha:]]+://)?"`

An additional 0.16% (22) of the patterns were excluded because they were empty or otherwise malformed so as to cause a parsing error.

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*.

3.5 Analyzing the corpus of regexes

3.5.1 Parsing feature tokens

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 KLE operator (specified using the asterisk `*`), the ADD operator (specified using the plus `+`), two CG elements (specified using pairs of parenthesis `(` and `)`), and the STR position (specified using the caret `^`). A detailed description of all studied features is provided in Section 1.3.

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

Table 3.2 Frequency of feature appearance in Projects, Files and Patterns, with number of tokens observed and the maximum number of tokens observed in a single pattern.

rank	code	example	% projects	nProjects	nFiles	nPatterns	nTokens	maxTokens.
1	ADD	`z+`	73.2	1,204	9,165	6,003	11,136	30
2	CG	`(caught)`	72.6	1,194	9,559	7,130	12,707	17
3	KLE	`.*`	66.8	1,099	8,163	6,017	11,620	50
4	CCC	`[aeiou]`	62.4	1,026	7,648	4,468	8,179	42
5	ANY	`.`	61.1	1,005	6,277	4,657	7,119	60
6	RNG	`[a-z]`	51.6	848	5,092	2,631	8,043	50
7	STR	`^`	51.4	846	5,458	3,563	3,661	12
8	END	`$`	50.3	827	5,393	3,169	3,276	12
9	NCCC	`[^qwxf]`	47.2	776	3,947	1,935	2,718	15
10	WSP	`\s`	46.3	762	4,704	2,846	6,128	32
11	OR	`a|b`	43	708	3,926	2,102	2,606	15
12	DEC	`\d`	42.1	692	4,198	2,297	4,868	24
13	WRD	`\w`	39.5	650	2,952	1,430	2,037	13
14	QST	`z?`	39.2	645	3,707	1,871	3,290	35
15	LZY	`z+?`	36.8	605	2,221	1,300	1,761	12
16	NCG	`a(?:b)c`	24.6	404	1,709	791	1,453	28
17	PNG	`(?P<name>x)`	21.5	354	1,475	915	2,399	16
18	SNG	`z{8}`	20.7	340	1,267	581	1,159	17
19	NWSP	`\S`	16.4	270	776	484	676	10
20	DBB	`z{3,8}`	14.5	238	647	367	573	11
21	NLKA	`a(?!yz)`	11.1	183	489	131	148	3
22	WNW	`\b`	10.1	166	438	248	408	36
23	NWRD	`\W`	10	165	305	94	149	6
24	LWB	`z{15,}`	9.6	158	281	91	107	3
25	LKA	`a(?=bc)`	9.6	158	358	112	133	4
26	OPT	`(?i)CasE`	9.4	154	377	231	238	2
27	NLKB	`(?<!x)yz`	8.3	137	296	94	117	4
28	LKB	`(?<=a)bc`	7.3	120	255	80	99	4
29	ENDZ	`\Z`	5.5	90	149	89	89	1
30	BKR	`\1`	5.1	84	129	60	73	4
31	NDEC	`\D`	3.5	58	92	36	51	6
32	BKRN	`(P?=name)`	1.7	28	44	17	19	2
33	VWSP	`\v`	0.9	15	16	13	14	2
34	NWNW	`\B`	0.7	11	11	4	5	2

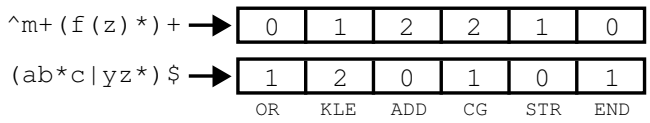

Figure 3.4 Two patterns parsed into feature vectors

3.5.2 Feature usage within the corpus

Table 3.2 displays feature usage from the corpus in terms of the number of patterns, files and projects, as well as in terms of tokens used.

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 as described in Section 1.3. The *example* column provides a short example of how the feature can be used.The next six columns contain usage statistics providing a variety of perspectives on how frequently the features are used in the observed population.

The *% projects* column contains the percentage of projects using a feature out of the 1,645 projects scanned that contain at least one pattern in the corpus. The *nProjects* column provides the number of projects that contain at least one usage of a feature. Assuming that one project generally corresponds to some high-level goal of a programmer or a team of programmers, these values provide a sense of how frequently a feature is *part of a software solution* in even the slightest way. Because of the generality of this measure and the goal of this study to gauge how features of regular expressions are used in general, these values are used to determine the rank of a feature.

The *nFiles* column specifies the number of files that contain at least one observed usage of the feature. For reference, recall that a total of 18,547 files were scanned that contain at least one feature usage. Assuming that programmers organize code into separate files based on what the code needs to do, this number can provide insight into the variety of different conceptually separate *task categories* a feature is used for.

The *nPatterns* column contains the number of patterns in which a feature was observed. Each regex is compiled from a particular pattern and performs at least one function desired by a programmer. Therefore the number of patterns composed using a feature can provide insight into the number of *specific tasks* a feature is used for.

The *nTokens* column, gives the total number of tokens observed for a feature, combining the token counts of all patterns in the corpus. This value provides a sense of how often the language feature is used *for any task*.

The last column, *maxTokens*, gives the maximum number of times that a feature appears in a single regex. Assuming that a feature that a programmer finds convenient is used more frequently in a given regex, this value provides a sense of *convenience* provided by the feature.

3.6 Feature Support

One issue that has persisted as a major pain point in the study of regular expressions is the lack of a concise summary comparing what features are supported in different regular expression language variants. This work provides such a summary in this section, and goes on to investigate what features are supported in reasoning tools for regular expressions. In the tables presented in this section the filled circle (●) means that a feature is supported, and the empty circle (○) means that a feature is not supported.

3.6.1 Caveats to consider when comparing feature sets

The variation among the supported feature sets is not easy to define. Often the same feature is essentially supported, but nuances exist so that the exact behavior of the feature still varies enough to have an effect on code that relies on regexes using that feature. One example of this is the OPT feature (e.g., `(?i)cAsE`), for which different engines have different sets of options. Python's set of 7 options is small compared to Tcl which has 15 or so. In Table 3.3 if the following 3 core options are supported: `(?ism)`, then the variant will be shown as having that feature. However in all other cases, to the best knowledge of the author, a strict view is taken when considering if two variants support the same feature - it should have the exact same syntax and behavior in order for the feature to be considered the same feature in two variants. Documentation of engines varies in detail and quality, so that often the particular behavioral details and full feature set is only known to developers of the engine. In this attempt to document some of the variations in feature support, no attempt is made to address these minor nuances and tricky details, but instead the focus is on documenting the presence or absence of

features at a high level. Most of the data presented here was determined by directly attempting to use a feature and noticing if either the engine threw an exception, or the expected effect was noticeably missing. This effort required hundreds of small experiments that will not be documented in detail at this time. A cursory treatment of where the information came from is provided in Section 3.6.2. These tables should not be relied upon in life-or-death situations, as some error is certainly possible. In such applications, a user may want to verify engine behavior using tests, consulting the documentation and source code as needed.

3.6.2 Choosing languages to compare feature support

Instead of using language popularity alone to determine what languages to include, these languages were selected to optimize for the intersection of variety of regular expression languages covered, and ease of testing feature inclusion. For example, Java and RE2 provide excellent and thorough documentation of their feature sets, and provide two entirely different variants. Although C and C++ are very popular languages, their regular expression libraries use external standards like ECMA (used by JavaScript) and POSIX ERE, and do not provide a distinct language of their own. For Python, Perl, Ruby, JavaScript and Java, testing a for a feature can be quickly accomplished in a browser or a terminal. For RE2, POSIX ERE and .Net no tests were performed, but documentation was good enough, and the language variants seem significant enough to try and include them. Two notably absent regular expression languages are the NSExpressions variant used by Apple in the Swift and Objective-C languages (no acceptably detailed documentation was found), and the well documented but wildly exotic syntax of Vim Regular Expressions which are very interesting but would unnecessarily inflate the size of the tables. So for 15 (75%) of the top 20 languages listed elsewhere, (i.e. not MATLAB, Swift, Objective-C, SQL or Assembly Language), the tables presented here should provide useful information.

3.6.3 Ranked feature support

Table 3.3 compares support for the 34 features studied in this thesis amongst Perl, Python, Ruby, .Net, JavaScript, RE2, Java and POSIX ERE (i.e., grep, sed, etc.). No languages share

the functionality of Python's ENDZ feature (preferring the LNLZ feature for that syntax). Only RE2 and Perl support Python-style named capture groups, and only Perl supports Python-style named back-references. JavaScript does not support options (OPT) or positive or negative look-backs (LKB, NLKB respectively). RE2 does not support any look-arounds (LKB, NLKB, LKA and NLKA) or back-references. POSIX ERE only supports 15 of the 34 studied features and Ruby does not support vertical whitespace (VWSP), but all remaining features are supported by all the other variants. The top nine features by rank are supported in all eight variants. These results support the relevance of the feature set selected for detailed study in this thesis. The implication here is that patterns written for one engine using this feature set are very likely to be interpreted the same way by other engines, which is good for portability.

3.6.4 Alien feature support

Table **??** describes feature support for a selection of 34 features (alien to the studied feature set) chosen from the eight languages being investigated. A reference code and small example are provided to aid in understanding. Several of these features actually represent an entire family of up to 12 features, like PXCC (e.g., `[:alpha:]`), EREQ (e.g., `[[=o=]]`), JAVM (e.g., `\p{javaMirrored}`), UNI and NUNI (e.g., `\pL` and `\PM`), but only one feature from such a family is selected for space considerations. Perl is notable for supporting the most features overall, and POSIX ERE is notable for supporting the smallest number of features.

3.6.5 Alien feature descriptions

The following brief descriptions of features alien to the studied feature set are provided to aide in understandability of Table 3.4. For a more detailed description, the reader will have to consult the documentation provided by a supporting variant.

RCUN: example: `(?n)` description: recursive call to group n

RCUZ: example: `(?R)` description: recursive call to group 0

GPLS: example: `\g{+1}` description: relative back-reference

GBRK: example: `\g{name}` description: named back-reference

GSUB: example: `\g<name>` description: Ruby-style subroutine call

KBRK: example: `\k<name>` description: .Net-style named back-reference

IFC: example: `(?(cond)X)` description: if conditional

IFEC: example: `(?(cnd)X|else)` description: if else conditional

ECOD: example: `(?{code})` description: embedded code

ECOM: example: `(?#comment)` description: embedded comments

PRV: example: `\G` description: end of previous match position

LHX: example: `\uFFFF` description: long hex values

POSS: example: `a?+` description: possessive modifiers

NNCG: example: `(?<name>X)` description: .Net-style named groups

MOD: example: `(?i)z(?-i)z` description: flag modulation (on and off anywhere)

ATOM: example: `(?>X)` description: atomic or possessive non-capture group

CCCI: example: `[a-z&&[^f]]` description: custom character class intersection

STRA: example: `\A` description: absolute beginning of input

LNLZ: example: `\Z"` description: end of input, or before last newline

FINL: example: `\z` description: absolute end of input, like ENDZ

QUOT: example: `\Q...\E` description: quotation

JAVM: example: `\p{javaMirrored}` description: java defaults

UNI: example: `\pL` description: Unicode defaults

NUNI: example: `\PS` description: Unicode negated defaults

OPTG: example: `(?flags:re)` description: flags just for inside this NCG

EREQ: example: `[[=o=]]` description: equivalence classes

PXCC: example: `[:alpha:]` description: POSIX defaults

TRIV: example: `[^]` description: trivial CCC, matches everything

CCSB: example: `[a-f-[c]]` description: custom character class subtraction

VLKB: example: `(?<=ab.+)` description: variable-width look-behinds. harder to implement

BAL: example: `(?<close-open>)` description: balanced groups (.Net version of recursion)

NCND: example: `(?(<n>)X|else)` description: named conditionals

BRES: example: `(?|(A)|(B))` description: branch numbering reset (A and B capture into the same group number)

QNG: example: `(?'name're)` description: single-quote named groups

3.6.6 Feature support in regex analysis tools

Tools for analyzing and reasoning about regular expressions are very attractive to language researchers, as well as industries that use regular expressions for mission critical applications (airlines, NASA, FBI, NSA, Oracle, etc.). The more features supported by an analysis tool, the further research and development can be advanced for all of these domains. On the other hand, the more features that developers of an analysis tool attempt to support, the more complex the implementation of the tool becomes, so at some point developers of an analysis tool are likely to make a judgment about whether or not to support a feature. In Table 3.5, the features supported by brics, hampi, Rex and Automata.Z3 are compared. The features of Rex in particular are of importance in this thesis, as Rex was used for the analysis described in Section link, and lack of support for various features was a major limiting factor for what could be included in the analysis.

What features each tool supports was determined in a variety of ways. For brics, the set of supported features was collected using the formal grammar[3]. For hampi, he set of regexes included in the test suite `lib/regex-hampi/sampleRegex` file within the hampi repository[4] were examined to determine which features hampi supports (this may have been an overestimation, as this included more features than specified by the formal grammar[5]). For Rex, the feature set was collected empirically when attempting to use Rex as described in Section link. For Automata.Z3, a file containing sample regexes[6] was examined to determine which features it

[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://github.com/AutomataDotNet/Automata/blob/master/src/Automata.Z3.Tests/SampleRegexes.cs

supports. This may be an underestimation, as the set of patterns provided is small. Hampi supports the most features (25 features), followed by Rex (21 features), Automata.Z3 (14 features) and brics (12 features). Rex and hampi support the 14 most commonly used features, whereas Automata.Z3 supports 11 of these features and bricks supports nine. No projects support the four look-around features LKA, NLKA, LKB and NLKB. Hampi supports named back-references, and no other back-reference support is available in any other tool. Hampi supports the LZY, NCG, PNG and OPT features, whereas brics, Automata.Z3 and Rex do not.

3.7 Discussion of utilization and feature analysis results

This section is not completed yet - intention is to review it all one more time when other sections are more complete and then write the discussion based on that read.

3.7.1 Implications

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

3.7.2 Opportunities for future work

3.7.2.1 Alternative techniques for building a corpus

any number of different regular expression languages like Perl Regular Expressions, Java Regular Expressions or .Net Regular Expressions. Also independent of the regular expression language, the regexes studied could come from a variety of sources like sourceforge, bitbucket, a private repository, or even from github using a different technique than the one used to build the corpus (described in Mention how exploring character details like literals, hex, octal and supported escape specials like bell, vertical wsp, etc is an opportunity for future work

ok for discussion of rationale for checking feature sets Similarly in JavaScript and POSIX ERE, the pattern `"a\Z"` compiles to a regex matching the string `"aZ"`, because the sequence `"\Z"` has no special significance and the backslash is ignored. In Python Regular Expressions, this sequence

does have significance - a feature matching the absolute end of the string (after the last newline). However, in Java, Perl, .Net and many other variants this sequence has a slightly different meaning (absolute end or before last newline). 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.

Unable to get enough information about Swift's underlying NSRegularExpression to include it in the table - a strong contender for future work! Also wanted to get Vim's features but do not have time, and it is a very alien feature set!.

3.7.3 Threats to validity

not good enough corpus, human error in testing lang table

Table 3.3 What regular expression languages support features studied in this thesis?

rank	code	example	Python	Perl	.Net	Ruby	Java	RE2	JavaScript	POSIX ERE
1	ADD	`z+`	●	●	●	●	●	●	●	●
2	CG	`(caught)`	●	●	●	●	●	●	●	●
3	KLE	`.*`	●	●	●	●	●	●	●	●
4	CCC	`[aeiou]`	●	●	●	●	●	●	●	●
5	ANY	`.`	●	●	●	●	●	●	●	●
6	RNG	`[a-z]`	●	●	●	●	●	●	●	●
7	STR	`^`	●	●	●	●	●	●	●	●
8	END	`$`	●	●	●	●	●	●	●	●
9	NCCC	`[^qwxf]`	●	●	●	●	●	●	●	●
10	WSP	`\s`	●	●	●	●	●	●	●	○
11	OR	`a|b`	●	●	●	●	●	●	●	●
12	DEC	`\d`	●	●	●	●	●	●	●	○
13	WRD	`\w`	●	●	●	●	●	●	●	○
14	QST	`z?`	●	●	●	●	●	●	●	●
15	LZY	`z+?`	●	●	●	●	●	●	●	○
16	NCG	`a(?:b)c`	●	●	●	●	●	●	●	○
17	PNG	`(?P<name>x)`	●	●	○	○	○	●	○	○
18	SNG	`z{8}`	●	●	●	●	●	●	●	●
19	NWSP	`\S`	●	●	●	●	●	●	●	○
20	DBB	`z{3,8}`	●	●	●	●	●	●	●	●
21	NLKA	`a(?!yz)`	●	●	●	●	●	○	●	○
22	WNW	`\b`	●	●	●	●	●	●	●	○
23	NWRD	`\W`	●	●	●	●	●	●	●	○
24	LWB	`z{15,}`	●	●	●	●	●	●	●	●
25	LKA	`a(?=bc)`	●	●	●	●	●	○	●	○
26	OPT	`(?i)CasE`	●	●	●	●	●	●	○	○
27	NLKB	`(?<!x)yz`	●	●	●	●	●	○	○	○
28	LKB	`(?<=a)bc`	●	●	●	●	●	○	○	○
29	ENDZ	`\Z`	●	○	○	○	○	○	○	○
30	BKR	`\1`	●	●	●	●	●	○	●	●
31	NDEC	`\D`	●	●	●	●	●	●	●	○
32	BKRN	`(P?=name)`	●	●	○	○	○	○	○	○
33	VWSP	`\v`	●	●	●	○	●	●	●	●
34	NWNW	`\B`	●	●	●	●	●	●	●	○

Table 3.4 What other features are supported in various languages?

code	example	Python	Perl	.Net	Ruby	Java	RE2	JavaScript	POSIX ERE
RCUN	`(?n)`	○	●	○	○	○	○	○	○
RCUZ	`(?R)`	○	●	○	○	○	○	○	○
GPLS	`\g{+1}`	○	●	○	○	○	○	○	○
GBRK	`\g{name}`	○	●	○	○	○	○	○	○
GSUB	`\g<name>`	●	●	○	●	○	○	○	○
KBRK	`\k<name>`	○	●	●	●	●	○	○	○
IFC	`(?(cond)X)`	○	●	●	○	○	○	○	○
IFEC	`(?(cnd)X|else)`	○	●	●	○	○	○	○	○
ECOD	`(?{code})`	○	●	○	○	○	○	○	○
ECOM	`(?#comment)`	●	●	●	●	○	○	○	○
PRV	`\G`	○	●	●	●	●	○	○	○
LHX	`\uFFFF`	○	●	●	●	●	○	●	○
POSS	`a?+`	○	●	○	●	●	○	○	○
NNCG	`(?<name>X)`	○	●	●	●	●	○	○	○
MOD	`(?i)z(?-i)z`	○	●	●	●	●	●	○	○
ATOM	`(?>X)`	○	●	●	●	●	○	○	○
CCCI	`[a-z&&[^f]]`	○	○	○	●	●	○	○	○
STRA	`\A`	●	●	●	●	●	●	○	○
LNLZ	`\Z`	○	●	●	●	●	●	○	○
FINL	`\z`	○	●	●	●	●	●	○	○
QUOT	`\Q...\E`	○	●	○	○	●	●	○	○
JAVM	`\p{javaMirrored}`	○	○	○	○	●	○	○	○
UNI	`\pL`	○	●	○	○	●	●	○	○
NUNI	`\PS`	○	●	○	○	●	●	○	○
OPTG	`(?flags:re)`	○	●	●	●	●	●	○	○
EREQ	`[[=o=]]`	○	○	○	○	○	○	○	●
PXCC	`[:alpha:]`	○	●	○	●	○	●	●	●
TRIV	`[^]`	○	○	○	○	○	○	●	○
CCSB	`[a-f-[c]]`	○	○	●	○	○	○	○	○
VLKB	`(?<=ab.+)`	○	○	●	○	○	○	○	○
BAL	`(?<close-open>)`	○	○	●	○	○	○	○	○
NCND	`(?(<n>)X|else)`	○	●	●	●	○	○	○	○
BRES	`(?|(A)|(B))`	○	○	○	○	○	○	○	○
QNG	`(?'name're)`	○	○	●	●	○	○	○	○

Table 3.5 What features are supported by regular expression analysis tools?

rank	code	example	brics	hampi	Rex	Automata.Z3
1	ADD	`z+`	●	●	●	●
2	CG	`(caught)`	●	●	●	●
3	KLE	`.*`	●	●	●	●
4	CCC	`[aeiou]`	●	●	●	●
5	ANY	`·`	●	●	●	○
6	RNG	`[a-z]`	●	●	●	●
7	STR	`^`	○	●	●	●
8	END	`$`	○	●	●	○
9	NCCC	`[^qwxf]`	●	●	●	○
10	WSP	`\s`	○	●	●	●
11	OR	`a|b`	●	●	●	●
12	DEC	`\d`	○	●	●	●
13	WRD	`\w`	○	●	●	●
14	QST	`z?`	●	●	●	●
15	LZY	`z+?`	○	●	○	○
16	NCG	`a(?:b)c`	○	●	○	○
17	PNG	`(?P<name>x)`	○	●	○	○
18	SNG	`z{8}`	●	●	●	●
19	NWSP	`\S`	○	●	●	○
20	DBB	`z{3,8}`	●	●	●	●
21	NLKA	`a(?!yz)`	○	○	○	○
22	WNW	`\b`	○	○	○	○
23	NWRD	`\W`	○	●	●	○
24	LWB	`z{15,}`	●	●	●	○
25	LKA	`a(?=bc)`	○	○	○	○
26	OPT	`(?i)CasE`	○	●	○	○
27	NLKB	`(?<!x)yz`	○	○	○	○
28	LKB	`(?<=a)bc`	○	○	○	○
29	ENDZ	`\Z`	○	○	○	●
30	BKR	`\1`	○	○	○	○
31	NDEC	`\D`	○	●	●	○
32	BKRN	`\g<name>`	○	●	○	○
33	VWSP	`\v`	○	○	●	○
34	NWNW	`\B`	○	○	○	○

CHAPTER 4. Equivalent representations of regex and their frequencies in the corpus

4.1 Defining five equivalence classes

4.1.1 Conceptual Overview

This chapter introduces possible refactorings in regular expressions by identifying equivalence classes of Python Regular Expressions, identifying what representations are possible in each equivalence class, and also identifying what transformations between representations are possible. As with source code, in regular expressions there are often multiple ways to express the same semantic concept. For example, `AAA*` matches two `A`s followed by zero or more `A`s. This matching behavior is identical to the behavior of the syntactically different regex `AA+`, which matches two or more `A`s. What is not clear is which representation, `AAA*` or `AA+`, is preferred.

Preferences in regular expression 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. By investigating which representation appears most frequently in source code, we can establish a community standard and suggest refactorings based on conformance to that standard.

Figure 4.1 displays five equivalence classes in grey boxes. 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`). This work will often use the term *group* as shorthand for 'equivalence class'.

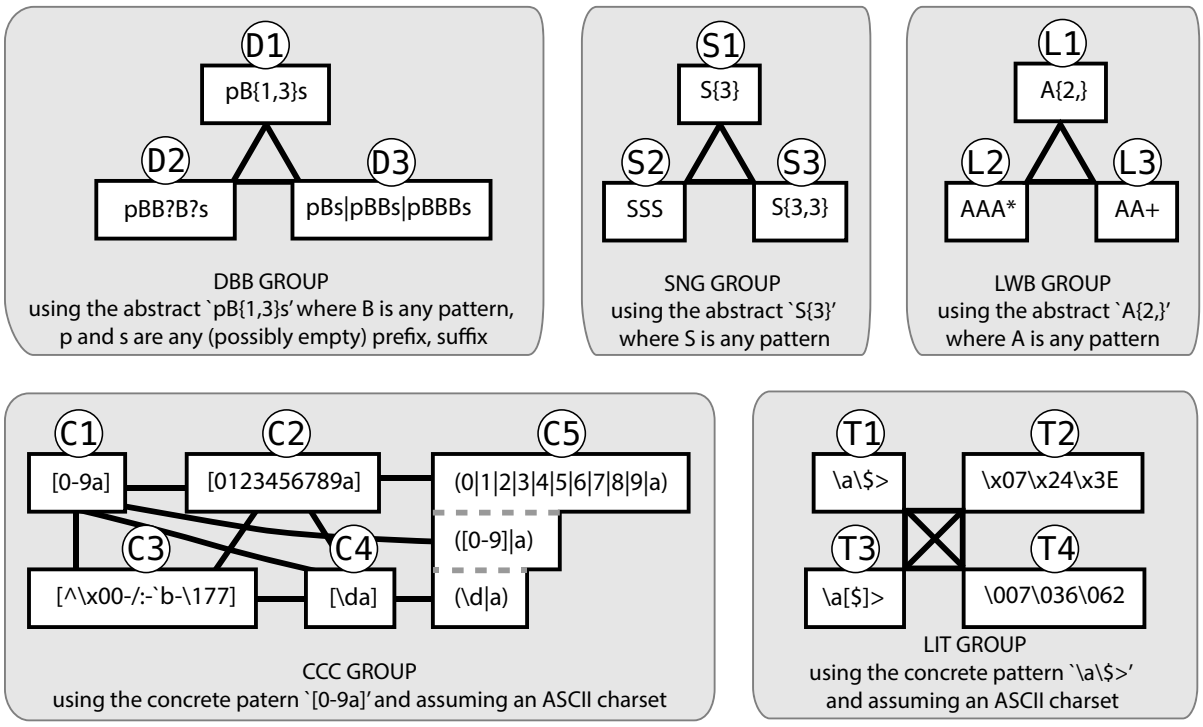

Figure 4.1 Equivalence classes with various representations of semantically equivalent representations within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

Each equivalence class has multiple *nodes* which each represent different ways to express or *represent* the behavior of a particular regex. Examples of various semantically equivalent *representations* of a regex are shown in white boxes. A *representation* is a particular regex that expresses matching behavior using the style of a particular *node*.

As an example of one equivalence class, consider the LWB group. Each node in the LWB group has a lower bound on repetitions. Regexes `A{2,}`, `AAA*` and `AA+` are semantically equivalent regexes belonging to the nodes L1, L2 and L3, respectively. The undirected edges between nodes define possible refactorings. Identifying the best direction for each arrow in the possible refactorings is discussed in Section **??**.

Figure 4.1 uses specific examples to more clearly illustrate the characteristics of each node. However, the `A's in the LWB group abstractly represent any element, and the number of elements is free to vary. We chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero. Next, we describe the characteristics of all nodes of each group in detail:

4.1.2 CCC Group

The Custom Character Class (CCC) group contains five nodes that each require the expression of a set of characters, as is typical when using the CCC feature. For example, the regex `b[ea]t` will match both `"bet"` and `"bat"` because, between the `` `b' `` and `` `t' ``, the CCC `[ae]` specifies that either `` `a' `` or `` `e' `` (but not both) must be present. 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 in Python Regular Expressions. Next, we provide descriptions of each node in this equivalence class:

C1: Any regex using the RNG feature in a CCC like `[a-f]` as shorthand for all of the characters between `` `a' `` and `` `f' `` (inclusive) belongs to the C1 node. C1 does *not* include any regex using the NCCC feature. All regex containing NCCC belong to the C3 node.

C2: Any regex that contains at least one CCC without any RNG or defaults belongs to the C2 node. For example, `[012]` is in C2 because it does not use any RNG or defaults, but `[0-2]` is not in C2 because it uses RNG. Similarly, `[Q\d]` is not in C2 because it uses the DEC default character class. Membership to C2 only requires one CCC without RNG or defaults, so `[abc][0-9\s]` *does* belong to C2 because it contains `[abc]`.

C3: Any regex using the NCCC feature belongs to the C3 node. For example `[^ao]` belongs to C3, and `[ao]` does not (notice the `` `^' `` character after the `` `[' ``).

For a given charset (e.g., ASCII, UTF-8, etc.), any CCC can be represented as an NCCC. Consider if the PRE engine was using an ASCII charset containing only the following 128 characters: `\x00-\x7f`. Consider that a CCC representing the lower half: `[\x00-\x3f]` can be represented by negating the upper half: `[^\x40-\x7f]`.

C4: Any regex using a default character class in a CCC like `[\d]` or `[\W]` belongs to the C4 node.

C5: Any regex containing an OR of length-one sequences (including defaults or other CCCs) belongs to the C5 node. These representations can be transformed into a CCC syntax by removing the OR operators and adding square brackets. For example `(\d|a)` in C5 is equivalent to `[\da]` in C4.

Because an OR cannot be directly negated, it does not make sense to have an edge between C3 and C5 in Figure 4.1, though C3 may be able to transition to C1, C2 or C4 first and then to C5.

A regex can belong to multiple nodes of the CCC group. For example, `[a-f\d]` belongs to both C1 and C4. The edge between C1 and C4 represents the opportunity to express the same regex as `[a-f0-9]` by transforming the default digit character class into a range. This transformed version would only belong to the C1 node. Not all regexes in C1 contain a default character class that can be factored out. For example `[a-f]` belongs to C1 but cannot be transformed to an equivalent representation belonging to C4.

4.1.3 DBB Group

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

D1: Any regex that uses the DBB feature (curly brace repetition with a different 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 regex that uses the QST feature (a question mark indicating zero-or-one repetition) belongs to D2. An example regex belonging to D2 is `zz?`, which matches `"z"` and `"zz"`.

When a regex belonging to D1 has zero as the lower bound, it can be transformed to a representation belonging to D2 by replacing the DBB feature and the element it operates on (like the `B{0,2}` in `pBB{0,2}s`) with n new regexes composed of the element operated on by DBB followed by QST, where n is equal to the upper bound in the DBB. For example `B{0,2}` has a zero lower bound and an upper bound of 2, so it can be represented as `B?B?`. Therefore `pBB{0,2}s` can become `pBB?B?s`.

D3: Any regex that uses OR to express repetition with different upper and lower boundaries like `pBs|pBBs|pBBBs` belongs to D3. The example `pB{1,3}s` becomes `pBs|pBBs|pBBBs` by explicitly stating the entire set of strings matched by the regex in an OR.

Note that a regex 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.

4.1.4 LIT Group

All regexes 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. Our examples use the ASCII charset, in which all characters can be expressed using hex codes like `\x3A` and octal codes like `\072`. The LIT 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. For example `a` belongs to T1.

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

T3: Any ordinary character wrapped in square brackets so that it becomes a CCC containing exactly one character belongs to T3. An example of a regex belonging to T3 is `[x][y][z]`. This style is used most often to avoid using a backslash so that a special character is treated as an ordinary character like `[|]`, which must otherwise be escaped like `\|`.

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

Patterns often fall in several of these representations. For example, `abc\007` includes literal elements `a`, `b`, and `c`, and also the octal element `\007`, thus belonging to T1 and T4.

4.1.5 LWB Group

The LWB group contains all regexes that specify only a lower boundary on the number of repetitions required for a match.

L1: Any regex using the LWB feature like `A{3,}` belongs to the L1 node. This regex will match `"AAA"`, `"AAAA"`, `"AAAAA"`, and any number of A's greater or equal to 3.

L2: Any regex using the KLE feature like `X*` belongs to the L2 node. The regex `X*` is equivalent to `X{0,}` because both will match zero or more `X` elements.

L3: Any regex using the ADD feature like `T+` belongs to the L3 node. The regex `T+`, which means one-or-more `` `T' ``s is equivalent to `T{1,}`.

Regexes can belong to multiple nodes in the LWB group. Within `A+B*`, the `A+` maps this regex to L3 and `B*` maps it to L2. Refactorings from L1 to L3, and L2 to L3 are not possible when the lower bound is zero and the regex is not repeated in sequence. For example neither `A{0,}` from L1, nor `A*` from L2 express behavior that can be represented using the ADD feature.

4.1.6 SNG Group

This equivalence class contains three nodes, each expressing SNG repetition in different ways.

S1: Any regex using the SNG feature like `S{3}` belongs to the S1 node. This example regex defines the string `"SSS"` where three `` `S' `` characters appear in a row.

S2: Any regex that is explicitly repeated two or more times and could use repetition operators belongs to the S2 node. For example `coco` repeats the smaller regex `co` twice and could be represented as `(co){2}`, so `coco` belongs to S2. Regex containing double letters like `foot` also belong to S2.

S3: Any regex with a double-bound in which the lower and upper bounds are same belongs to S3. For example, `S{3,3}` specifies a string where `` `S' `` appears a minimum of 3 and maximum of 3 times, which is the string `"SSS"`.

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).

4.1.7 Example regex

Regexes 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]`. Moving from L2 to L1 would mean replacing `[^ ]*` with `[^ ]{0,}`, resulting in a refactored regex of: `[^ ]{0,}\.[A-Z][A-Z][A-Z]`.

4.2 Counting representations in nodes

4.2.1 Overview of the node counting process

This work finds defines a community standard for each equivalence class by counting the number of regexes in each node. A program was implemented that iterates through the corpus once for each of the 18 nodes, adding regexes to sets that represent nodes based on the definitions in Section 4.1. A regex is a *candidate* for membership in a node if it is possible for that regex belong to a node. For six nodes, the presence of a feature is enough to determine membership without ambiguity. For four nodes, the presence of a feature and a search of the regex's pattern is enough to determine candidacy for membership. The remaining eight nodes require more advanced *filters* to determine candidacy for membership.

To verify accuracy and obtain a final node count, all sets were dumped to text files and reviewed manually. Regexes that had been erroneously added to a node were removed. The regexes that had not been added to any node in a given equivalence class were also dumped to a file, and manually searched for regexes that belonged to some node but had been erroneously filtered out. This process was iterated on several times to refine the filters used in the

implementation. Even after many iterations, manual verification was still required for D3 and others?. The final outcome of this node counting process is summarized in Table 4.1.

The source code used to perform the node counting process is available on Github[1]. The following sections provide implementation details sufficient to recreate the process.

4.2.2 Artifact details

The implementation is written in Java 8, and depends on antlr 3.5.2 because the PCRE parser used[2] uses antlr to identify the features present in a pattern. Implementation details about building the corpus can be found in Section **??**. For this experiment, the corpus was re-loaded from a text file. Tests verify that this re-loaded corpus is identical to the corpus built from scratch.

Each regex in the corpus contains the original pattern used to compile the regex, a `org.antlr.runtime.tree.` (parse tree) created by the PCRE parser from the pattern, and a set of integers used to identify the set of Python projects that used this regex in at least one utilization.

4.2.2.1 Tokenstreams

A *tokenstream* is a string that can be generated from a regex's parse tree to represent the parsed regex as a string. Unlike the pattern used to compile the regex, all ambiguities due to multiple meanings of characters, balanced parenthesis, etc. have already been resolved by the parser. This sequence of tokens is still a context free language with nested, balanced `DOWN` and `UP` tokens, therefore it cannot be fully described or parsed using Java Regular Expressions (which do not support the recursion feature or similar). However, the tokenstream can be searched for necessary conditions when constructing filters to identify candidacy for node membership, and regexes that are erroneously added to a node can be removed manually, as described in Section 4.2.1.

Each leaf of the parse tree is assigned a text representation based on the token names used by the parser. The 'bullet' character was chosen as a delimiter that is not present in any text

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

representation. Invisible characters and Unicode characters like the 'bullet' are represented using a hex representation of their bytes. The tokenstream is created by joining the text representation of each leaf with the delimiter, as shown in FigureN.

4.2.3 Implementation details of determining node membership

4.2.3.1 Membership based only on feature presence

The nodes which only require a check for the presence of a feature to determine membership are described here:

D2 requires QST (zero-or-one repetition using question mark)

S1 requires SNG (curly braces with one number inside specifying the number of repetitions)

L1 requires LWB (curly braces with one number followed by a comma, specifying a lower bound on repetition)

L2 requires KLE (kleene star indicating zero-or-more repetitions)

L3 requires ADD (plus character indicating one-or-more repetitions)

C3 requires NCCC (a negated custom character class, where a `^' negates a CCC like `[^X]`)

4.2.3.2 Membership based on a feature presence and search of the pattern

S3 requires DBB, and requires the regex's pattern to match `\{(\d+),\1\}` which guarantees that both bounds of DBB are the same by capturing the first bound in `(\d+)` and then back-referencing the captured number.

T2 requires the presence of some hex character representation in the pattern, which is verified by searching the regex's pattern with the regex `\\x[a-f0-9A-F]{2}`.

T4 requires the presence of some octal character representation in the pattern, which is verified by searching the regex's pattern with the regex `((\\0\d*)|(\\\d{3}))`. Python-style octals require 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`.

D3 requires OR (alternation using the `|`), and requires the regex's pattern to match `(?<=[|])([^ \\]+)\1+|`. The core of this regex is `([^ \\]+)\1+` which describes a string that contains a repetition of some sequence at least two times. The sequence is captured by `([^ \\]+)` and then back-referenced by `\1+`, which can appear one or more times as specified by ADD.

In this regex, the lookbehind `(?<=[|])` and lookahead `(?=[|])` match when the `|` character is found to the left or right of the core regex, respectively. The regex used as a filter matches either `(?<=[|])([^ \\]+)\1+` or `([^ \\]+)\1+(?=[|])`. All patterns in the corpus that have some repeated sequence as an alternative within an OR match this regex. For example the pattern `"a|aa"` compiles to the regex `a|aa` which belongs to D3. This filter produced a list of 113 candidates which were narrowed down manually to 10 actual members.

The space and slash characters are excluded from the sequence described by `([^ \\]+)` in order to exclude common false positives like `"some text  |more text"` and `"\\|/"`, which match the regex but do not belong to D3. Note that these false positives were manually verified as described in Section 4.2.1, ensuring that in the corpus used, no valid members of D3 were excluded. However it is possible for this filter to exclude a regex with a pattern like `"(  | )"` or `"(\\\\|\\)"` which should belong to D3.

4.2.3.3 Membership based on a feature presence and filters

D1 requires DBB (curly braces with two numbers separated by a comma inside), where the two numbers are different. When these two numbers are the same, then the regex cannot be refactored within the DBB group, but instead belongs to the S3 node. It is likely impossible to directly specify that two numbers are different in Java Regular Expressions, but it is possible to identify when two numbers are the same using back-references, and eliminate only these. So the same regex used to identify members of S3: `\{(\d+),\1\}` was used to find all such same-bound parts of a pattern and replace them with `"{$1}"`, where `$1` is referencing the digit captured by `(\d+)}`. In effect this step is actually performing a refactoring from S3 to S1. All patterns that still contain DBB syntax must

belong to D1, so the modified pattern is then searched using `\{\d+,\d+\}` to determine membership in D1.

TODO 7 MORE finish the remaining 7 filter implementation descriptions

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.

T1 requires that no characters are wrapped in brackets or are hex or octal characters, which matches over 91% of the total regexes analyzed.

T3 requires that a single literal character is wrapped in a custom character class (a member of T3 is always a member of C2).

C1 requires that a non-negative character class contains a range.

C2 requires that there exists a custom character class that does not use ranges or defaults.

C4 requires the presence of a default character class within a custom character class, specifically, `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`.

C5 requires an OR of length-one sequences (literal characters or any character class).

4.3 Node counting results

For each node, Table 4.1 presents the number of regexes belonging to that node, and the number of projects containing at least one such regex belonging to that node. The *node* column references the node labels (like 'T1') in Figure 4.1. The *description* column briefly describes the rules for node membership, followed by an *example* regex from the corpus. The *nRegexes* column counts the regexes that belong to a given node, followed by the percent of regexes out of 13,597 (the total number of regexes in the corpus). The *nProjects* column counts the projects that contain a regex belonging to the node, followed by the percentage of projects out of 1,544 (the total number of projects scanned that contain at least one regex from the corpus). Recall that the regexes of the corpus are all unique and could appear in multiple projects, hence the project support is used to show how pervasive the node is across the whole community. For example, 2,479 of the regexes belong to the C1 representation, representing 18.2% of regexes in

Table 4.1 How frequently is each alternative expression style used?

Node	Description	Example	nRegexes	% regexes	nProjects	% projects
C1	CCC using RNG	`^[1-9][0-9]*$`	2,479	18.2%	810	52.5%
C2	CCC listing all chars	`[aeiouy]`	1,903	14.0%	715	46.3%
C3	any NCCC	`[^A-Za-z0-9.]+`	1,935	14.2%	776	50.3%
C4	CCC using defaults	`[-+\d.]`	840	6.2%	414	26.8%
C5	CCC as an OR	`(@|<|>|-|!)`	245	1.8%	239	15.5%
D1	repetition like {M,N}	`^x{1,4}$`	346	2.5%	234	15.2%
D2	zero-or-one repetition	`^http(s)?://`	1,871	13.8%	646	41.8%
D3	repetition using OR	`^(Q|QQ)\<(.+)\>$`	10	.1%	27	1.7%
T1	not in T2, T3 or T4	`get_tag`	12,482	91.8%	1,485	96.2%
T2	has HEX like `\xF5`	`[\x80-\xff]`	479	3.5%	243	15.7%
T3	wrapped chars like [$]	`([*]|[:])`	307	2.3%	268	17.4%
T4	has OCT like `\0177`	`[\041-\176]+:$`	14	.1%	37	2.4%
L1	repetition like {M,}	`(DN)[0-9]{4,}`	91	.7%	166	10.8%
L2	kleene star repetition	`\s*(#.*)?$`	6,017	44.3%	1,097	71.0%
L3	additional repetition	`[A-Z][a-z]+`	6,003	44.1%	1,207	78.2%
S1	repetition like {M}	`^[a-f0-9]{40}$`	581	4.3%	340	22.0%
S2	sequential repetition	`ff:ff:ff:ff:ff:ff`	3,378	24.8%	861	55.8%
S3	repetition like {M,M}	`U[\dA-F]{5,5}`	27	.2%	32	2.1%

the corpus. These appear in 810 projects, representing 52.5%. Regexes belonging to D1 appear in 346 (2.5%) of the regexes in the corpus, but only 234 (15.2%) of the projects. In contrast, 39 *fewer* regexes are in node T3, but 34 *more* projects use regexes from T3, indicating that D1 is more concentrated in a few projects and T3 is more widespread across projects.

4.4 Discussion

4.4.1 Preferences indicated by community support

Using the count of regexes in each node provided in Table 4.1, the most preferred nodes for each group are C1, D2, T1, L2, and S2. In this section the practical issues of refactoring between nodes are explored and several preferences between nodes are identified.

4.4.1.1 Community based CCC refactoring

C1 may be preferred overall because ranges are shorter Within the CCC group, C1 has the most regexes (2,479), suggesting that there may be a preference to write a regex

with a range whenever possible. This makes sense, since a range shortens the regex, and programmers are often trying to make their code as short and efficient as possible.

These three regexes from the corpus belong to C2: `i[3456]86`, `[Hh][123456]` and `-py([123]\.[0-9])$`. The community preference for regexes to use C1 suggests refactorings to `i[3-6]86`, `[Hh][1-6]` and `-py([1-3]\.[0-9])$` respectively.

C2 contains few sequential character sets, so it is hard to refactor out of On inspection, there are very few such regexes in C2 that are obvious candidates for refactoring to C1. This is true because most of regexes in C2 do not express ranges of characters, but instead express non-continuous sets. The following regexes (or regex fragments) extracted from the corpus illustrate this point: `[?/:|]+`, `coding[:=]`, `([\\"]|[^\ -~])`, `^[012TF\*]{9}$`, `\?|[-+]?[.\w]+$`.

Refactoring out of C3 is generally awkward Similarly, very few regexes in C3 actually seem like candidates for refactoring to C1 on inspection. Although the transformation is possible, most regexes in C3 seem to be negating just one or two characters like `[^:]*:` and `^([^/:]+):`. Refactoring these to C1 exposes an awareness of the charset and uses ranges that often start or end with invisible characters. For example these two regexes when refactored to C1 (assuming ASCII) would be `[\x00-9;-\x7F]*:` and `^([\x00-.0-9;-\x7F]+):`. The most notable candidate for refactorings going out of C3 is probably from C3 to C4, because many NCCC simply represent the negated version of some default character class. For example the NCCC `[^a-zA-Z0-9_]` appears in 8 regexes belonging to C3, and could be refactored to `[\W]` which belongs to C4. However, according to community standards the preferred representation may be in C3, not C4.

Refactoring out of C4 may be recommended Refactorings going from C4 to C1 are possible for the DEC and WRD default character classes, (i.e., `[\d]` to `[0-9]` and `[\w]` to `[0-9a-zA-Z_]`) and may be recommendable based on the standards of the community observable in Table 4.1. Similarly refactorings from C4 to C3 are possible for the negative default character classes (i.e., `[\D]` to `[^0-9]` and `[\W]` to `[^0-9a-zA-Z_]`). Refactorings

from C4 to C2 might make sense regarding the WSP default character class (i.e., `[\s]` to `[ \t\r\n\v\f]`), but on inspection most regexes in C2 that are close to this new regex typically omit the `\v` and `\f` characters, with `\r` and `\n` also omitted at times.

Refactoring from C5 to C2 is always recommended Regexes belonging to C5 are the most proportionally widespread compared to other members of the CCC group, with about as many regexes (245) as there are projects that they appear in (239). One interpretation of this is that these regexes are not pulled from other projects, but are original compositions in each project. All of the regexes belonging to C5 could be refactored to C2, which offers a more preferred representation style according to the community. Three such possible refactorings from the corpus are: `(a|b)*?c` to `[ab]*?c`, `:|\*|\?|"|<|>|\\|` to `[:*?"<>\\"]` and `^(?:!|&|\\*)$` to `^(?:[!&*])$`.

4.4.1.2 Community based DBB refactoring

D1 to D2 is recommended for small upper bounds There are about six times as many regexes in D2 (1,871) than there are in D1 (346). D3 has only 10 regexes and appears in only 27 projects, so it is clearly not recommended according to community standards. Refactoring from D1 to D2 is always possible, and always recommended by the community standard. For example the regex `^[\n\r]{0,1}` from the corpus belonging to D1 becomes `^[\n\r]?`. This is a simple change in syntax because the upper bound is low. Similarly the corpus regex `(\d{2,3})` becomes `(\d\d\d?)` when transformed from D1 to D2.

In contrast the corpus regex `^.{3,20}$` belongs to D1, and can be converted to the equivalent representation in D2: `^....?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?$`. This new regex is not as compact and seems ridiculous. Intuitively, it does not make any sense to recommend a refactoring that explodes a small regex into a huge one. One reason that D2 may have more community support is that the use case of specifying zero-or-one of something is a very natural idea, and may occur more frequently than the need to specify a particular range using DBB.

4.4.1.3 Community based LIT refactoring

T3 to T1 is always recommended It is not surprising that most regex belong to the T1 node, since ordinary characters are necessary for most string specifications. Regex like `[$][{]\d+:([^}]+)[}]` belong to T3 and seem to be trying to avoid escaping special characters by wrapping them in a CCC. This regex can be refactored to T1 yielding `\$\{\d+:([^}]+)\}`, which is recommended due to overwhelming community support of T1.

T2 and T4 may be special cases The most popular regex from T4 `[\041-\176]+:$` appears in 13 projects. The character class defined in this regex represents the printable ASCII characters. This could be refactored to the equivalent `[!-~]$` in T1, but the original regex may offer more intuition about the size of the range being specified.

Similarly, the most popular regex from T3 `[\x80-\xff]` appears in 81 projects and refers to a range of characters above ASCII. This representation may offer some useful intuition about the range being specified, so a refactoring to T1 is not recommended at this time. More study is needed into the readability of this type of range and the alternative using T1.

T4 to T2 is always recommended It is not always possible to use a literal character to specify a character. For characters that cannot be represented directly, a refactoring from T4 to T2 is always recommended. T2 has more than 34 times as many regexes (479) as T4 (14) and so based on community standards, all of these should be refactored.

4.4.1.4 Community based LWB refactoring

L2 to L3 may be recommended L2 has 6,017 regexes while L3 has 6,003 and so they are very closely tied in terms of number of regexes. In terms of projects, L3 has a slight advantage with 1,207 compared to the 1,097 containing some L2 regex. This indicates a slight preference for L3 over L2, but is not a strong indicator. Furthermore a refactoring from L2 to L3 requires an additional repeated element in the sequence to be present before the element to which KLE is applied. For example the regex belonging to L2 `kk*` has this extra `` `k' `` that can be used to transform this regex into a regex belonging to L3: `k+`. However the

regex `k*` does not have another `k', so no transformation is possible. Patterns of the 6,017 regexes belonging to L3 were searched using the regex `([^\\)\]])\1*`, locating 38 patterns where their corresponding regexes could be transformed this way. The most popular example (8 projects) was the regex `(?::*)` which would become `(?:+)`.

L1 to L3 is recommended for lower bounds L1 has only 91 regexes or almost 67 times fewer than L3, so the community supports a refactoring to L3. The most popular regex in L1 (32 projects) is `\n{2,}` which can be transformed to `\n\n+` belonging to L3. The regex in L1 with the largest lower bound is `[1-9A-HJ-NP-Za-km-z]{26,}\\Z` which when transformed to a regex in L3 would become too long to typeset in this thesis, and that refactoring cannot be recommended. However only 3 regexes had a lower bound greater than 6 and only 10 had a lower bound greater than 4, so most regexes in L1 are good candidates for refactoring to L3.

4.4.1.5 Community based SNG refactoring

refactoring to S2 may be recommended for small repetitions Inspecting the contents of S2 reveals that most of these regexes belong to S2 because they contain normal words like `"session"` or `"https"` that happen to have a repetition of characters. This artificially inflates the size of S2 and presents a challenge when trying to use the community standards rationale to suggest a refactoring to S2 from S1 or S3. For example, consider the most popular (32 projects) regex from S1: `^[a-f0-9]{40}$`. Expanding this out so that it uses sequential repetition would create a very long regex and cannot be recommended. However transforming the regex `^(-?\d+)(\d{3})` from S1 yields the regex `^(-?\d+)(\d\d\d)` which may be recommended by the community standards, but more investigation is needed.

refactoring out of S3 is recommended S3 only has 27 regexes, and all of them seem to be abusing the DBB feature by making the upper and lower bounds identical. Perhaps these regexes started off with different bounds and were fixed as time went on to have identical bounds. Due to the inflation of S2, it is not clear whether to recommend a refactoring to S1

or S2. Perhaps the best recommendation is to refactor low numbers of repetitions of small elements to S2, and all others to S1.

4.4.2 Opportunities for future work

Equivalence Class Models This study uses 5 equivalence classes, each with 3 to 5 nodes. These equivalence classes are very inclusive of regexes with very different behavior, and are defined largely by the features used by a regex. Future work could look into much more narrowly defined equivalence models, specific to particular behaviors of the most frequently observed regexes. For example a node could require the presence of a very specific, frequently used CCC like `[a-zA-Z0-9_-]` which can be refactored to `[\w-]`.

In addition to breaking the five equivalence classes into more granular nodes, future work could model refactorings outside of these groups. Due to the functional variety and significant number of features to consider, this work does not provide a list of all possible refactoring groups. However the following 5 additional equivalence classes are examples of other possible groups:

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

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

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

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

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

Future work could also use reasoning tools for regular expressions, like Microsoft's Automata library to identify populations of equivalent regexes with differing representations, even identifying *all* equivalence classes present in a given corpus.

Regex Programming Standards Many organizations enforce coding standards in their repositories to ease understandability. Using an equivalence class model and the node counting technique described in this chapter could help to objectively develop regular expression standards for a given development community like Mozilla or OpenBSD.

Studying a different corpus A technique similar to the one described in this work could be applied to a different corpus. Ideas about alternative ways to build a corpus of regexes can be found in Section 3.7.2.1). The concept of a community standard would be reinforced by regexes sourced from a very specific community like only text editor projects, or only shopping cart frameworks.

4.4.3 Threats to validity

Because the technique of determining a node count includes manual verification, is possible that some regexes were not removed from a node that should have been removed, or were included when they should not have been. This does not represent a serious threat, however, because a small number of errors would not significantly change the main results of the work.

The rules used to define the nodes of equivalence classes may have been too permissive, or may have not been designed in the best possible way, resulting in an experiment that fails to detect some other more interesting refactoring that was not considered. As the first work on regular expression refactoring, this outcome is unavoidable and is not considered a major problem.

Because of the way the corpus is randomly selected from Github the projects it references may be biased towards homeworks and small pet projects, or frequently cloned projects like the linux kernel. This threat is not of significant concern because no claims are made about what community is being represented, and the concept of community is not being used strictly to determine the refactoring recommendations, but is more of a guide.

CHAPTER 5. Comprehension of regex representations

5.1 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, it should be possible to infer which representations are more desirable and which are more smelly. This study was implemented on Amazon's Mechanical Turk with 180 participants. Each regex was evaluated by 30 participants. The regexes used were designed to belong to various nodes of the equivalence class graphs depicted in Figure 4.1.

5.1.1 Metrics

The understandability of regexes was measured using two complementary metrics, *matching* and *composition*.

Matching: Given a regex and a set of strings, a participant determines which strings will be matched by the regex. There are four possible responses for each string, *matches*, *not a*

Table 5.1 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

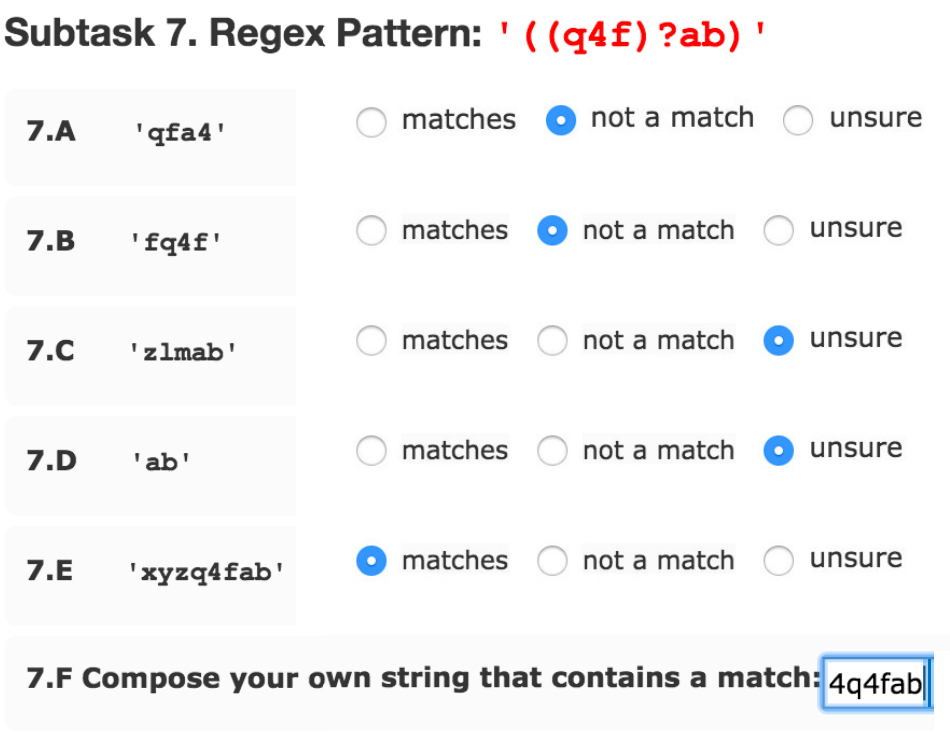

Figure 5.1 Example of one HIT Question

match, *unsure*, or blank. An example from our study is shown in Figure 5.1. The use of the term 'matches' in this chapter is consistent with the meaning described in Section 1.2.2 - if any substring of a target string belongs to the set of strings specified by a particular regex, then that regex is said to *match* that target string. For ease of expression, a string is said to *match* a regex if that regex matches the string.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example, consider regex `RR*` and five strings shown in Table 5.1, 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. Response with all blank or unsure responses are referred to as an 'NA'. Out of 1800 questions, 1.8%(32) were NA's (never more than 4 out of 30 per regex).

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

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

5.1.2 Implementation

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

5.1.2.1 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 regexes mean: `a+`, `(r|z)`, `\d`, `q*`, and `[p-s]`. To pass the qualification, workers had to answer four of the five questions correctly.

5.1.2.2 Selecting pairwise comparisons

Using the regexes in the corpus as a guide, ten metagroups were created for this study. The first six metagroups (re-numbered for simplicity) each contain three pairs of regexes, as shown in the following six lists:

Metagroup 1: testing S1 vs S2

S1	`%([0-9A-Fa-f]{2})`	S1	`&d([aeiou]{2})z`	S1	`fa[lmnop]{3}`
S2	`%([0-9a-fA-F][0-9a-fA-F])`	S2	`&d([aeiou][aeiou])z`	S2	`fa[lmnop][lmnop][lmnop]`

Metagroup 2: testing C1 vs C4, focusing on DEC

C1	`([0-9]+)\.([0-9]+)`	C1	`xg1([0-9]{1,3})%`	C1	`[a-f]([0-9]+)[a-f]`
C4	`(\d+)\.(\d+)`	C4	`xg1(\d{1,3})%`	C4	`[a-f](\d+)[a-f]`

Metagroup 3: testing C1 vs C4, focusing on WRD

C1	`&([A-Za-z0-9_]+);`	C1	`1q[A-Za-z0-9_][A-Za-z0-9_]`	C1	`[tuv[A-Za-z0-9_]]`
C4	`[&(\w+);]`	C4	`[1q\w\w]`	C4	`[tuv\w]`

Metagroup 4: C4 vs (C3 or C2), covering the other defaults

C3	`[^0-9A-Za-z]`	C3	`[^0-9]`	C2	`[\t\r\f\n ]`
C4	`[\W_]`	C4	`[\D]`	C4	`[\s]`

Metagroup 5: testing L2 vs L3 (note that the pair on the left is not equivalent, due to an oversight - the first regex was meant to be `\.\.*`)

L2	`\..*`	L2	`zaa*`	L2	`RR*`
L3	`\.+`	L3	`za+`	L3	`R+`

Metagroup 6: testing T1 vs T3

T1	`(\$\{)\d+(:[^}]+\})`	T1	`t\.\$+\d+\*`	T1	`\{\$(\d+\.\d)\}`
T3	`([$][{])\d+(:[^}]+[}])`	T3	`t[.][$]+\d+[*]`	T3	`[{][$](\d+[.]\d)[}]`

Four additional metagroups were created, each containing two sets of three equivalent regexes. These groups of three were useful because more comparisons could be drawn from the same number of experiments. The regexes used are shown in the following four lists:

Metagroup 7: testing D1 vs D2 vs D3

D1 `((q4f){0,1}ab)`	D2 `((q4f)?ab)`	D3 `(q4fab|ab)`
D1 `(dee(do){1,2})`	D2 `(deedo(do)?)`	D3 `(deedo|deedodo)`

Metagroup 8: testing C1 vs C2 vs C5

C1 `no[w-z]5`	C2 `no[wxyz]5`	C5 `no(w|x|y|z)5`
C1 `tri[a-f]3`	C2 `tri[abcdef]3`	C5 `tri(a|b|c|d|e|f)3`

Metagroup 9: testing C2/T1 vs C5/T1 vs C2/T4 (provides T1 vs T4 and C2 vs C5)

C2/T1 `([}{])`	C5/T1 `(\{|\})`	C2/T4 `([\072\073])`
C2/T1 `([:;])`	C5/T1 `(:|;)`	C2/T4 `([\0175\0173])`

Metagroup 10: testing C1/T2 vs C1/T4 vs C2/T1 (provides only T2 vs T4)

C1/T2 `xyz[\x5b-\x5f]`	C1/T4 `xyz[\0133-\0140]`	C2/T1 `xyz[_\[\]`\^\\]`
C1/T2 `t[\x3a-\x3b]+p`	C1/T4 `t[\072-\073]+p`	C2/T1 `t[:;]+p`

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

Although this design provides 42 pairwise comparisons (18 from pairs, 24 from triples), six comparisons had to be dropped due to a design flaw since the regexes performed transformations from multiple equivalence classes. For example `([\072\073])` is in C2 and T4. This regex was paired with `(:|;)` 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. Also, the first pair of regexes from metagroup 5: `\..*` and `\.+` are not equivalent. The first regex was meant to be `\.\.*`. Data gathered for this pairing was ignored.

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 comparisons 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 35 pairwise comparisons across 14 edges from Figure 4.1.

5.1.2.3 Composing Tasks

For each of the 26 groups of regexes, five strings were created, where at least one matched and at least one did not match. These strings were used to compute the matching metric.

Once all the regexes and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a regex from each of the 10 metagroups. Randomize the order of these 10 regexes, as well as the order of the matching strings for each regex. After adding a question asking the participant to compose a string that each regex 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, the five matching strings and the space for composing a string is shown in Figure 5.1.

5.1.2.4 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.

5.2 Population characteristics

5.2.1 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

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 5.2 Participant Profiles, $n = 180$

Table 5.2 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	80%	60%	0.001	87%	37%	**¡0.001**
E2	D2 – D3	2	78%	87%	**0.011**	88%	97%	0.085
E3	C2 – C5	4	85%	86%	0.602	88%	95%	**0.063**
E4	C2 – C4	1	83%	92%	**0.075**	60%	67%	0.601
E5	L2 – L3	2	86%	91%	0.118	97%	100%	0.159
E6	D1 – D2	2	84%	78%	0.120	93%	88%	0.347
E7	C1 – C2	2	94%	90%	0.121	93%	90%	0.514
E8	T2 – T4	2	84%	81%	0.498	65%	52%	0.141
E9	C1 – C5	2	94%	90%	0.287	93%	93%	1.000
E10	T1 – T3	3	88%	86%	0.320	72%	76%	0.613
E11	D1 – D3	2	84%	87%	0.349	93%	97%	0.408
E12	C1 – C4	6	87%	84%	0.352	86%	83%	0.465
E13	C3 – C4	2	61%	67%	0.593	75%	82%	0.379
E14	S1 – S2	3	85%	86%	0.776	88%	90%	0.638

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 5.2 summarizes the self-reported participant characteristics from the qualification survey.

5.3 Matching and composition comprehension results

5.3.1 Analysis

For each of the 180 HITs, a matching and composition score was computed for each of the 10 regexes, using the metrics described in Sectionn. Since 30 separate participants responded to five string matching problems and one composition problem for each of the 60 regexes, there were 30 independent understandability evaluations for each representation. An average of 0.53 out of 30 of these responses were NAs per regex, with the maximum number of NAs being four. These 26-30 independent matching scores for each regex were used to determine if an understandability preference exists for each of the 36 pairwise comparisons.

For example, one group had regexes `RR*` and `R+`, which represents a transformation between L2 and L3. The former had an average matching score of 86% and the latter had an

Table 5.3 Equivalent regexes with a significant difference in readability

node	regex	match	compose	refactoring	node	regex	match	compose
T4	`([\072\073])`	66%	50%	$\overrightarrow{T4T1}$	T1	`([:;])`	81%	87%
T4	`([\0175\0173])`	54%	23%		T1	`([}{])`	79%	87%
D2	`((q4f)?ab)`	79%	83%	$\overrightarrow{D2D3}$	D3	`(q4fab|ab)`	85%	97%
D2	`(deedo(do)?)`	77%	93%		D3	`(deedo|deedodo)`	90%	97%

average matching score 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. The other pairwise comparison performed between L2 and L3 group used the pair `zaa*` and `za+`. Considering both of these regex pairs, the *overall matching score* for the regexes belonging to L2 was 0.86 and the *overall matching score* for L3 was 0.91. The *overall composition score* for L2 was 0.97, with 1.00 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 5.2, with this specific example appearing in the E5 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 display the average composition scores for the representations and the relevant p-value, also using the Mann-Whitney test of means.

Table 5.2 presents the results of the understandability analysis. A horizontal line separates the top two edges from the bottom 12. In E1 and E2, 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 defined in this study. Specifically, $\overrightarrow{T4T1}$ and $\overrightarrow{D2D3}$ are likely

Table 5.4 Average Unsure Responses Per Pattern By Node (fewer unsures are lower)

Node	Number of Patterns	Unsure Responses Per Pattern
T4	4	8.5
T2	2	5.5
T3	3	2.7
T1	3	2.7
D2	2	2.5
C3	2	2
C5	4	2
D1	2	2
C4	9	1.9
S1	3	1.7
S2	3	1.7
L2	3	1.3
C1	8	1
C2	5	1
D3	2	1
L3	3	0.7

to improve understandability. The specific nodes, regexes, matching scores and compositions scores that led to these refactoring suggestions are shown in Table 5.3

Participants were able to select *unsure* when they were not sure if a string would be matched by a regex (Figure 5.1). From a comprehension perspective, this indicates some level of confusion and is worth exploring.

For each regex, the number of responses containing at least one unsure was observed, representing confusion when attempting to answer matching questions for that regex. The regexes were then grouped into their representation nodes and an average number of unsures was computed per regex. For example, four regexes belonged to C5 and the number of unsures for those regexes was: 2,3,3 and 0 so the average number of unsures for C5 was 2. A higher number of unsures may indicate difficulty in comprehending a regex 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 5.2.

These nodes and their average number of unsure responses are organized in Table 5.4. These results strongly corroborate the refactorings suggested by the understandability analysis

for both the LIT group (i.e., $\overrightarrow{T4T1}$) and the DBB group (i.e., $\overrightarrow{D2D3}$) because both refactorings go from nodes with more unsures to nodes with fewer unsures (T4 has 8.5 whereas T1 has 2.7, and D2 has 2.5 whereas D3 has 1). The one regex from T4 that had the most unsures of any regex (i.e., 10 out of 30) was `xyz[\0133-\0140]`. The regex with the lowest composition score (7 out of 30) and matching score (0.54) was `([\0175\0173])`, which only had 6 unsures.

5.4 Discussion of comprehension results

5.4.1 Implications

Two statistically significant refactorings $\overrightarrow{T4T1}$ and $\overrightarrow{D2D3}$ were identified by the results presented in Table 5.2. A detailed view of the results for these refactorings is presented in Table 5.3. The first refactoring, $\overrightarrow{T4T1}$, makes sense because the octal syntax is far more exotic and difficult to understand than plain characters. Composition improves notably from 23% for `([\0175\0173])` to 87% for `([}{])`. This results seems likely to generalize, as there is no reason to think that participants were less familiar with octal than programmers in general.

The second refactoring $\overrightarrow{D2D3}$, reduces confusion caused by the QST feature, by expanding the entire set of strings specified by the regex into an OR. The OR feature is fundamental to regular expressions, and so the regexes in D3 are very straightforward - essentially lists of strings, whereas the QST repetition may take a little thought. This result seems likely to generalize for very simple examples like the one that was tested, using only one QST operator.

This refactoring is not likely to scale, however, because a slightly more complicated regex like `a?b*(cd)?e?` would expand to the very long regex `ab*cde|b*cde|ab*e|b*e|ab*cd|b*cd|ab*|b*` which introduces the new challenge of visually parsing and remembering eight strings.

Although not statistically significant within the chosen alpha (0.05), all tested refactorings *out* of C2 (into C5, C4 and C1) provided at least a slight advantage on average in both matching and composing scores. This may not indicate a refactoring (suggesting what node to choose), but instead indicate that C2 is a smelly node.

The most notable difference in measured understandability is between `[\t\r\f\n ]` from C2 with a matching score of 83%, and `[\s]` from C4 with a matching score of 92%.

Moving from C2 to C5 is not as clear cut, with examples supporting both directions. For the regex `([:;])`, the matching score increased from 81% to 94% when moving to `(:|;)` However for `tri[abcdef]3`, the matching score decreased from 93% to 86% when moving to `tri(a|b|c|d|e|f)3`.

Moving from C2 regex `no[wxyz]5` with a matching score of 87% to either C1 or C5 boosted the matching score to 94% or 93%, respectively.

5.4.2 Opportunities for future work

easy section

Discuss the backslash explosions and readability issues!

5.4.3 Threats to validity

many - bad behavior by MT participants is possible. Also three noticeable design flaws: the mistake with `\..*`, the mistakes with pairings of nodes that could not be used (bc study was implemented before equiv class design was completed), and that some edges are not covered - example? Also these results are fairly think and the signal is not very loud, with the exception of T4 to T1. A more perfectly designed study, with a better set of participants could really build on what was done here - could do it better. However, the results shown seem to be valid, if quite limited.