Usage And Refactoring Studies Of Python Regular Expressions

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 occupation. I'd also like to thank my wife Chien Wen Hung and our cat Siva for practical and moral support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

CHAPTER 1. OVERVIEW

1.1 Introduction

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 the 1960's, no one has ever formally studied how they are used in practice, or how to apply refactoring principals to improve understandability and conformance to community standards. This thesis presents the original work of studying a sample of regexes taken from Python projects mined from GitHub, determining what features are used most often, defining some categories that illuminate common use cases, and identifying areas of significance for language and tool designers. 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.

1.2 Outline

1.2.1 Sections of this thesis

This thesis begins with an introduction detailing the research questions explored and the main contributions, followed by a section on related work, touching on historical milestones and applications of regular expressions, as well as work on mining repositories and refactoring that has similarities to this work. The next chapter provides the formatting standards and terminology used in this thesis to aide in understanding. Next, the five studies conducted

to explore the five research questions are each presented with their own separate discussion section that focuses on the results that particular study. Then a final discussion highlights the most important implications from each study, also presenting any implications gathered from the combination of results from multiple studies. Opportunities for future work are presented next, followed by a conclusion, an appendix of artifacts and a bibliography.

1.2.2 Research questions

Although regex have provided an essential search functionality for software development for half a century, 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, these five questions were formulated to begin the work of filling this fundamental knowledge gap. The following section articulates the motivations behind the questions explored in this thesis.

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

The features that allow regex users to compactly represent sets of strings are what power regular expressions. Gathering fundamental statistics about what features are used can inform many other issues in regular expression research, such as language and tool design.

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

If a well-fitting categorization scheme for regex behavior can be devised, these categories can provide insight into what users are really doing with regexes and in turn, what behaviors are most important for future regex technologies.

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

Experienced professional software developers are an excellent resource when studying real-world development practices. By asking developers about their regex usage patterns and preferences, the findings of other studies can be corroborated or brought under greater scrutiny.

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

There 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, what representations are chosen?

1.2.2.5 RQ5: Within five equivalence classes, what representations are more comprehensible?

Regexes in source code must be understood in order to be properly maintained, but regexes can be hard to understand. If the most understandable representation for a particular class of regular expressions can be determined, then the understandability of regexes can be increased through refactoring, easing the burden on maintainers.

1.3 Contributions

The contributions of this work are:

- An empirical analysis of regex feature usage in 13,597 regexes extracted from 1,645 open-source Python projects (Section 4.1.4),

- A comparison of supported features across eight regular expression languages, as well as a comparison of features supported by four regular expression analysis tools (Section 4.1.5),

- An approach for measuring behavioral similarity of regular expressions(Section 4.2.1.2), and qualitative analysis of clusters formed using that behavioral similarity measure (Section 4.2.3.2),

- A survey of 18 professional software developers about their habits, preferences and challenges using regular expressions (Section 4.3.2),

- Identification of equivalence classes for regular expressions with possible transformations within each class (Section 4.4.1),

- An empirical study of how frequently regexes are represented within equivalence classes, identifying refactoring opportunities based on these frequency measurements (Section 4.4.3),

- An empirical study with 180 participants evaluating the understandability of representations within equivalence classes, identifying refactoring opportunities based on these understandability measurements (Section 4.5.2),

- An evidence-based discussion of opportunities for future work in supporting programmers who use regular expressions, including refactoring regexes based on a variety of metrics, providing regex search functionality, migration support between languages, and fundamental research extending the techniques pioneered in this work (Section 6).

list several major results here

CHAPTER 2. BACKGROUND

2.1 Formatting And Feature Acronyms

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`. Pattern fragments used to represent feature tokens (usually to provide an example of what an acronym means), appear in parenthesis with a light grey background, like (`[^...]`).

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, and a single slash in a regex appears as `\\` where the pattern in source code is `"\\\\"`. Invisible characters such as newline will be represented within strings in gray, like `"first line.\nsecond line."`.

This thesis discusses the features used by regular expressions in depth. To facilitate this discussion, every feature is assigned an acronym composed of two to four capitol letters. For example, the Kleene star feature, (`*`), representing zero or more of some element, is referred to using the acronym 'KLE'. A concise presentation of the 34 features this thesis focuses on is presented in Table 4.2. A detailed description of all these features is provided in Appendix A. All other features mentioned in this thesis, but not studied in detail are briefly described in Appendix A.

2.2 Terminology

2.2.1 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 differentiate between particular variants (e.g., as used in the sentence: '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 *feature tokens*. For example the pattern `"a*"` is parsed into the (`a`) ordinary character token, and the (`*`) KLE token. A valid sequence of feature tokens 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 Appendix A.

2.2.2 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. For ease of expression, a string is said to *match* a regex if that regex matches the string.

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 return a `MatchObject`. Instead, the definition of *match* used in this thesis 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 any function provided by any engine to take action based on that match.

2.2.3   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.

The difference between regexes and patterns is important primarily when considering portability. Not all of the patterns used to compile the Python regexes studied in this thesis will necessarily compile to regexes in other languages. A discussion of what features are supported by different languages is provided in Section 4.1.5.

Most patterns compile to a functionally identical regex in most languages Examples have been shown of patterns that can have alternate meanings, or 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 multiple variants. The extent of feature overlap among variations is explored in Section 4.1.5, especially Table 4.5 and Table 4.6.

CHAPTER 3.   RELATED WORK

3.1   Milestones In Regular Expression History

3.1.1   Kleene's theory of regular events

In 1943, a new model for how nets of nerves might 'reason' to react to patterns of stimulus was proposed [McCulloch and Pitts (1943)]. In 1951, Kleene further developed this model with the idea of 'regular events' [Kleene (1951)]. In his terminology, 'events' are all inputs on a set of neurons in discrete time, a *definite event* is some explicit sequence of events, and a *regular event* is defined using three operators: 1. logical OR (|), 2. concatenation and 3. KLE (*) repetition 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 [Kleene (1956)].

3.1.2   First regex compiler

Many additional formalisms were built on Kleene's set of three operators, and then in 1967 Ken Thompson filed a patent [at Bell Labs (1971)] and published a paper [Thompson (1968)] for his implementation of the first regular expression compiler. This compiler was written in IBM 7090 assembly for a version of 'qed'[1] (quick editor) at Bell labs. Existing editors were only able to search and replace using whole words. Thompson's compiler enabled qed to search and replace using regular expressions. As described in his paper, Thompson's compiler accepted Kleene Regular Expressions and ordinary characters as input. Later versions of this compiler

---

[1] https://www.bell-labs.com/usr/dmr/www/qed.html

eventually supported the new features STR (`^`), END (`$`), ANY (`.`), CCC (`[...]`), RNG (`[a-z]`) and NCCC (`[^...]`). Although these features provided a useful shorthand, and could be considered a new language, whatever could be expressed using these features could also be expressed using only Kleene Regular Expressions features [Hopcroft et al. (2006)].

3.1.3 Early regular expressions in Unix

Thompson went on to create Unix in 1974 with Dennis M. Ritchie [Ritchie and Thompson (1974)]. Early Unix relied on '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 [McIlroy (1987)]. 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 [Hume (1988)]. This new language added the features CG (`(...)`), SNG (`a{1}`), DBB (`a{1,3}`), LWB (`a{1,}`), QST (`a?`), ADD (`a+`) as well as 12 default character classes similar to DEC (`\d`), but using syntax like (`[:digit:]`). This new language also introduced the 'backreference' BKR (`(a.b)\1`) feature. This feature, which goes 'back' and 'references' the content of a capture group, 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 [Hopcroft et al. (2006)]. Aho also wrote fgrep, which is optimized for efficiency instead of expressiveness using the AhoCorasick algorithm [Aho and Corasick (1975)].

3.1.4 Maturity of standards

In 1979, Hopcroft and Ullman published the 'Cindarella' textbook covering automata and theory supporting the syntax of grep (excluding back-references) [Hopcroft and Ullman (1979)]. Perl 2 was released in 1988 with some regular expression support [per (2015)], and included shorthand for default character classes like (`\d`) for DEC. The Perl community significantly boosted the popularity and user base of regular expressions [per (2001)].

In 1994, IEEE released the POSIX.2 standard [IEE (1994)], detailing specifications for shells and utilities, formally specifying the Basic Regular Expressions (BRE) and Extended Regular Expressions (ERE) languages. In 1997, the O'Reilly book 'Mastering Regular Expressions' [Friedl (2006)] was first published, providing tutorials on regular expression usage in plain language. In 1999, Henry Spencer released POSIX.2-compliant `regcomp` [spe (2015)], which is a regular expression library for C, as part of 4.4BSD Unix. By the time Perl 5.10 was released in 2007 [per (2016)], many advanced features had been introduced like recursion, conditionals and subroutines.

3.2   Applications Of Regex

A variety of applications for regular expressions is explored in this section.

3.2.1   End user applications

Find and replace utility   The task for which regular expressions were first implemented is finding and replacing strings in blocks of text. This remains a central application for regular expressions, which programmers can use to save effort. Utilities provided by text editors such as Emacs, Notepad++, Sublime Text and Eclipse include: incremental find and replace within a single file, batch find and replace within a single file or group of files, highlighting matched text and counting the number of matching strings. Emacs also provides utilities that delete all lines or retain all lines containing a match, and aligning columns by a regex-defined delimiter.

System administration   System administrators and power users rely on command-line utilities to accomplish complex computing tasks, often dealing with file names and the contents of configuration files. The output of one utility can be used as the input for another utility using *pipes*, and the mini-programs written using piped commands often rely on regular expressions to filter or transform strings in one step or another. For example, `grep` [gre (2015)] allows users to find strings that match a regex, and `sed` [sed (2004)] provides a replace functionality. The `find` [fin (2016)] utility searches filenames based on a regex, and tools like `git` [git (2015)]

and `cron` [cro (2016)] use regexes written in configuration files ('.gitignore' and 'crontab', respectively) to specify sets of files or sets of dates and times.

Searching fields in relational databases The popular SQL query language [Chamberlin and Boyce (1974)] uses the 'LIKE' operator and an exotic regular expression syntax ((`%`) for zero or more of any character, equivalent to `.*` , (`_`) to match any character, equivalent to `.` , and typical character classes using brackets) to search fields for strings. Modern relational database systems have expanded this syntax considerably with functions such as `REGEXP_LIKE` in Oracle [Ora (2003)], `REGEXP` in MySQL [MyS (2016)], `$regex` in MongoDB [Mon (2016)] and `regexp_replace` in PostrgreSQL [Pos (2016)].

3.2.2 Research and industry applications

Meta-programming Regular expressions are central to YACC and Lex, which are critical compiler tools for generating parsers used in the compilation process and lexing source files, respectively. In the case of YACC, regex are used as a meta-programming language specifying the behavior of a parser [Johnson (2006)]. Similarly in Lex, regexes are used to specify the behavior of a source code lexer [Lesk (2006)].

Regular expressions have also been used for test case generation [Ghosh et al. (2013); Galler and Aichernig (2014); Anand et al. (2013); Tillmann et al. (2014)], and as specifications for string constraint solvers [Trinh et al. (2014); Kiezun et al. (2013)]. Some data mining frameworks use regular expressions as queries (e.g., [Begel et al. (2010)]).

Network administration and security Regular expressions are used to encode forwarding paths in software-defined networks in 'Merlin' [Soulé et al. (2014)], and are used in network intrusion detection [network (2015); Sommer and Paxson (2003)] and deep packet inspection [Kumar et al. (2006); Yu et al. (2006)].

Regexes are employed in MySQL injection prevention [Yeole and Meshram (2011)] and in more diverse applications like DNA sequencing alignment [Arslan (2005)] or querying RDF

data [Lee et al. (2010); Alkhateeb et al. (2009)]. Efforts have also been made to expedite the processing of regular expressions on large bodies of text [Baeza-Yates and Gonnet (1996)].

3.2.3 Regex composition and analysis tools

Composition tools   Until this work (Chapter 4.5), regular expression understandability has not been studied directly, though prior work has suggested that regexes are hard to read and understand since there are tens of thousands of bug reports related to regular expressions [Spishak et al. (2012)]. Due in part to their common use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust regex creation [Spishak et al. (2012)] or to allow visual debugging [Beck et al. (2014)]. Other tools allow users to compose regular expressions using natural language[2].

Tools have also been developed to make regexes easier to understand, and many are available online. Some tools will, for example, highlight parts of regexes that match parts of strings as a tool to aid in comprehension.[3] Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [Babbar and Singh (2010); Li et al. (2008)].

Analysis tools   Research tools like Hampi [Kiezun et al. (2013)], and Rex [Veanes et al. (2010)], and commercial tools like brics [Møller (2010)] all support the analysis of regular expressions in various ways. Hampi was developed in academia and uses regular expressions as a specification language for a constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in applications such as test case generation [Anand et al. (2013); Tillmann et al. (2014)]. Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation. Automata.Z3[4] is one of a suite of tools developed by Microsoft to analyze regular expressions.

---

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

3.3 Similar Research

3.3.1 Mining and surveys for language feature analysis

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

3.3.2 Refactoring and smells

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

In software, code smells have been found to hinder understandability of source code [Abbes et al. (2011); Du Bois et al. (2006); Hermans (2016)]. Once removed through refactoring, the code becomes more understandable, easing the burden on the programmer. In regular expressions, such code smells have not yet been defined, perhaps in part because it is not clear what makes a regex smelly.

Code smells in object-oriented languages were introduced by Fowler [Fowler (1999)]. Researchers have studied the impact of code smells on program comprehension [Abbes et al. (2011); Du Bois et al. (2006)], finding that the more smells in the code, the harder the com-

prehension. This is similar to the work in this thesis, except we aim to identify which regex representations can be considered smelly. Code smells have been extended to other language paradigms including end-user programming languages [Hermans et al. (2012, 2014); Stolee and Elbaum (2011, 2013)]. The code smells identified in this work are representations that are not common or not well understood by developers. This concept of using community standards to define smells has been used in other refactoring literature for end-user programmers [Stolee and Elbaum (2011, 2013)].

CHAPTER 4. STUDIES

4.1 Usage And Support Of Regex Features

The primary goal of this experiment was to answer the question, 'How are regex used in practice, especially what features are most commonly used?'. Python was chosen because its regular expression feature set seemed to contain a good balance between having some advanced features, and not having too many rare features (this assumption was confirmed, as discussed in Section 4.1.5.2.). 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 4.1.1

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 Appendix A. Once the data about utilizations had been collected, some questions about the utilizations themselves were explored. This exploration can be read about in Section 4.1.1.

The patterns obtained from the utilizations were parsed using a PCRE parser to create Table 4.3. This table summarizes the findings of this experiment, that is, for each feature

<div align="center">

function   pattern   flags

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

</div>

Figure 4.1   Example of one regex utilization

described in Appendix A, this table shows the number of projects using that feature in some pattern (as well as other data). These findings are presented in Section 4.1.4.

The knowledge of how frequently each feature is used can provide context when comparing the sets of features supported by various regular expression analysis tools and language variants besides Python Regular Expressions. An exploration of 68 features (34 ranked by this study, and 34 other unranked features) is in Section 4.1.5. Finally a discussion of the impact of this study and threats to validity is in Section 4.1.6.

4.1.1   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.

4.1.1.1   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 4.1 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 4.1. As described in Section 2.2.1, a pattern specifies a 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 4.1 .

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 2.2.3, 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).

4.1.1.2   Omission of calls to compiled regexes

Every utilization recorded using the technique described in Appendix A is an invocation directly using the `re` library, like `re.compile(...)` or `re.search(...)`. However, this technique does not record calls on compiled objects. For example the regex described in Section 4.1.1.1 is stored in the variable `r1`, and a function call on that variable like `r1.search("-45")` is not recorded. However, this omission only impacts the interpretation of Figure 4.3, which describes which function calls were observed (calls to compiled objects are excluded). This issue does not impact the coverage of patterns, because every compiled regex like `r1` comes from a call to `re.compile(...)`, which is captured by the technique used in this study.

4.1.1.3   Selecting projects to mine for utilizations

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. In order to obtain utilizations from a pseudo-random, broad selection of projects, 3,898 projects containing Python code were mined for utilizations as described in Appendix A. 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: *repoID*. At the time the mining for utilizations used in this study was performed, the largest repoID was between 32 million and 33 million. Dividing these repoIDs 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 repoIDs 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 Appendix A, 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.

4.1.1.4   Observed utilizations of the re module

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 4.1.

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 4.1, 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.

---

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

Table 4.1   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

Flags and functions   As shown in figure   4.2, 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 4.3 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.
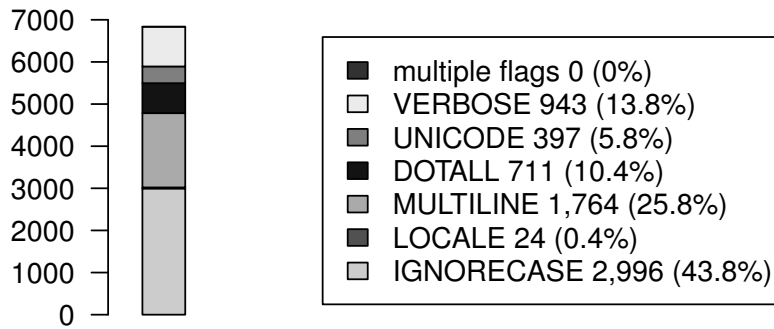
Figure 4.2   Which behavioral flags are used?
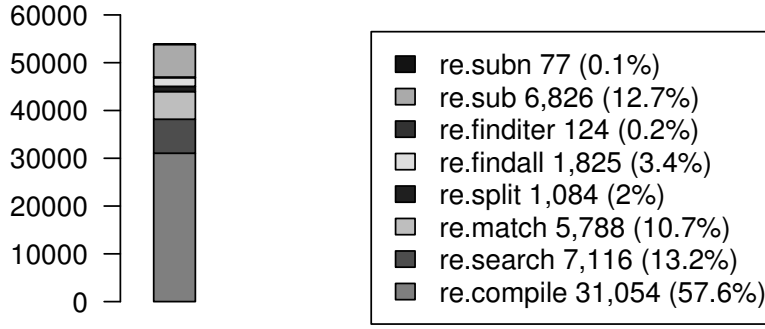
Figure 4.3   Which behavioral flags are used?

4.1.2   Building the corpus

4.1.2.1   Selecting a body of patterns from a set of utilizations

Patterns with behavioral flags and variables are excluded   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 behavioral flags (default and debug do not affect engine behavior) 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).

Normalizing patterns   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.

4.1.2.2   Parsing Python Regular Expression patterns using a PCRE parser

All studied features are recognizable   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 4.5, 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.

---

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

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.

Excluded patterns   This parser was unable to support 0.53% (73) of the patterns due to unsupported Unicode characters. Another 0.12% (17) of the patterns used PCRE features not valid in Python. Two additional patterns used the commenting feature, ECOM, which is valid in Python but is excluded to keep the analysis more succinct. An additional 0.16% (22) of the patterns were excluded because they were empty or otherwise malformed so as to cause a parsing error. In total, these excluded patterns represent 0.83% (114) of the 13,711 distinct patterns obtained. Excluded patterns are listed in Appendix A.

Corpus defined   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*.

4.1.3   Analyzing the corpus of regexes

4.1.3.1   Studied feature set

This thesis will focus on the features and syntax described in Table 4.2. Every regex in the corpus only uses features from this feature set. A detailed introduction to the functionality of these features as studied in this thesis is provided in Appendix A. The features of Python Regular Expressions analyzed 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 Appendix A.

4. Positions refer to a position between characters. They make assertions about the string on one or both sides of their position.

Table 4.2   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 ith 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

4.1.3.2   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 4.4. The pattern "^m+(f(z)*)+" contains four different types of tokens. It has the KLE operator (*), the ADD operator (+), two CG elements ((...)), and the STR position (^).

```
^m+(f(z)*)+  →   0  | 1  | 2  | 2  | 1  | 0
(ab*c|yz*)$  →   1  | 2  | 0  | 1  | 0  | 1
                OR   KLE  ADD  CG   STR  END
```

Figure 4.4   Two patterns parsed into feature vectors

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.

4.1.4   Frequency of feature usages

Table 4.3 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 (all features are described in Appendix A). 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.

222

25

Table 4.3   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

rank	language	library	variants provided by library
1	Java	java.util.regex	Java Regular Expressions
3	C++	std::regex	POSIX BRE & ERE & Awk, ECMAScript
4	C#	System.Text.RegularExpressions	.Net Regular Expressions
5	Python	re module	Python Regular Expressions
6	PHP	PCRE core extension	PCRE
7	Visual Basic .NET	System.Text.RegularExpressions	.Net Regular Expressions
8	JavaScript	RegExp object (built-in)	ECMAScript
9	Perl	perlre core library	PCRE
10	Ruby	Regexp class (built-in)	Ruby Regular Expressions
11	Delpi	RegularExpressions unit	PCRE
14	Swift	NSRegularExpression	NS Regular Expressions
15	Objective-C	NSRegularExpression	NS Regular Expressions
16	R	grep (built-in)	TRE, PCRE
17	Groovy	java.util.regex	Java Regular Expressions
18	MATLAB	regexp function (built-in)	MathWorks Regular Expressions
19	PL/SQL	LIKE operator (built-in)	SQL Regular Expressions
20	D	std.regex	D Regular Expressions

Table 4.4   Top 17 programming languages containing support for regexes, ranked according to popularity by tiobe.com, with the regular expression libraries built into them and the variants that they support.

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.

4.1.5   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 section provides such a summary, 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 (cavats about comparing features are described in Appendix A).

4.1.5.1   Libraries providing regular expression functionality

For most popular programming languages, various regular expression functions are provided using standard libraries or are built into the language. Table 4.1.5 describes the standard regular expression libraries or built-ins *provided as a core language feature* for all but three of the top 20 most popular languages ordered according to the TIOBE[3] index. The C, Visual Basic and Assembly Language languages (ranks 2, 12 and 13, respectively) do not provide built-in regular expression support and so are not shown. Alternative libraries are discussed in Appendix A.

4.1.5.2   Ranked feature support

Table 4.5 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.) as determined through consulting documentation and performing experiments. More details about the techniques used to decide if a feature is supported by a language or not are discussed in Appendix A. The rationale for selecting these languages is discussed in Appendix A.

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.

The studied feature set is representative These results support the relevance of the feature set selected for detailed study in this thesis, and the selection of Python for this

---

[3]http://www.tiobe.com/tiobe_index

Table 4.5   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	●	●	●	●	●	●	●	○

investigation. 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. Portability was discovered to be a pain point for developers, as discussed in Section 4.3.2.4.

4.1.5.3   Unranked feature support

Table 4.6 describes feature support for a selection of 34 *unranked* features (not in 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. A brief explanation of the functionality of these features is available in Appendix A

4.1.5.4   Feature support in regex analysis tools

Tools for analyzing and reasoning about regular expressions are very attractive to language researchers, and may have industry applications for critical systems. The more features supported by an analysis tool, the more regexes it can analyze. 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.

At some point developers of an analysis tool will need to choose a feature set to support. In Table 4.7, the features supported by brics, hampi, Rex and Automata.Z3 are compared. Details about how feature support was determined are provided in Appendix A. 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.

Table 4.6   What features, not studied in this thesis, 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 4.7 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	○	○	○	○

4.1.6   Discussion of feature analysis results

4.1.6.1   General Implications

Many of the implications of this work will not be apparent from the beginning, because of the fundamental nature of the investigation. However, some implications can be drawn directly from this data.

Defining a set of regex features that generalizes   As shown in Table 4.5, the feature set used in this study (except for PNG, NBKR and ENDZ) generalizes well across all feature-rich modern variants. This has implications for coding standards, and for software development where portability is a concern.

Providing a reference for language and tool designers   The largest implication for language and tool designers is that now, if they need to know what features are supported, or how frequently a feature is used, such information is available in a concise format.

For developers of the re module   Specific details about utilizations of the `re` module, such as function and flag usage frequency, and saturation within GitHub projects, has implications for the developers of the `re` module. Namely, the near-irrelevance of the 'locale' flag, the general trend toward compiling objects (presumably avoiding magic strings), and the apparent misconception that only one flag can be used at a time (the documentation clearly states that a bitwise-or of flags is effective[4], but this was never observed in over 50,000 utilizations).

4.1.6.2   Interpretations of feature frequency values

The following discussion uses the five interpretations for values from Table 4.3, as mentioned in Section 4.1.4.

- nProjects indicates how frequently a feature is *part of a software solution.*
- nFiles indicates the number of conceptually separate *task categories.*
- nPatterns indicates the number of *specific tasks* a feature is used for

---

[4]https://docs.python.org/2/library/re.html

- nTokens indicates how often the language feature is used *for any task*

- maxTokens provides a sense of *convenience* provided by the feature

KLE vs ADD In terms of patterns and tokens, these two features have very similar frequencies. Yet in terms of projects and especially files, ADD is used more often. KLE, however, appears 50 times in a single regex, compared to a max of 30 times for ADD. Given the assumptions listed above, this implies that ADD is effective in a broader range of task categories, but KLE is more convenient than ADD when its use is appropriate.

This makes sense, because KLE is so frequently used along with ANY as `.*` to consume 'all the chars, if any exist' - this provides users with a very general-purpose idiom that can be added to the beginning or end of the core pattern, or within a capture group, with little thought.

ADD on the other hand, specifies requirements that will cause a failure to match if they are not met. And that functionality is useful in more task categories than KLE, presumably because it specifies positively what to match, not just what is allowed but not required.

CG may be the real #1 In terms of files, patterns and tokens, CG is the most frequently used feature. This is easy to understand as the logical grouping it provides is natural and necessary for many tasks. CG is not the top ranked feature in terms of projects (by only 0.6%), but this could be explained as background noise, as the general trend is that CG is used most often in other categories. Surprisingly, it is not used more than 17 times in any pattern. This suggests that its use takes extra care (compared to KLE or ANY)

Clone smells in file-to-pattern ratio RNG appears in 5,092 files but is present in only 2,631 patterns. NCCC has about the same 2-to-1 ratio of files to patterns (3,947 to 1,935). Using the assumptions about what these numbers mean, a larger ratio of files to patterns should suggest that a feature is more limited in terms of what it can do (patterns relate to specific tasks), but these limited functions have a a larger variety of applications (files relate to categories of task). This implies that regexes containing features with a high file-to-pattern

ratio are more likely to be clones. Other features with a high file-to-pattern are OR (3,926 to 2,102) and WRD (2,952 to 1,430).

WNW is very convenient, for a position The STR, END, ENDZ, LKA, LKB, NLKB and NLKA features indicate zero-width positions in a string. Perhaps because they are more restrictive or complex than other features, their maxTokens values are lower than usual (at 12, 12, 1, 4, 3, 4 and 3 respectively). The exception to this rule is WNW with 36 appearances in a single pattern.

4.1.6.3   Threats to validity

Compared to the overall number of Python projects in existence, the number of projects used in this study is small. It may not be representative of Python projects on a whole. This is mitigated by the pseudo-random nature of selecting projects based on an arbitrary unit of division, several hundred thousand repository creation events apart (Section 4.1.1.3).

As discussed thoroughly in Section 2.2.3, patterns are not regexes, and there is always a risk that the patterns used to build the corpus would not port to other languages, making this analysis limited in application to only Python programmers. This risk is mitigated by Table 4.5, which proves a high level of portability for all regexes studied, and therefore some guarantee of applicability across languages.

Human error may have invalidated some feature presence/absence entries in Table 4.5. This is mitigated by the choice of languages, selected to make initial testing relatively fast, and re-testing by outside parties also relatively fast.

4.2   Categories Of Regex Usage Tasks

4.2.1   Clustering design

Regular expression languages are infinite and exhibit substantial variety, but programmers are likely to use them for a limited number of purposes. The goal of this study is to answer the question, 'What behavioral categories can be observed in regex?' so that designers of regular expression languages and end-user tools can better support what is most useful to programmers.

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

	A	B
A	1.0	0.9
B	0.5	1.0
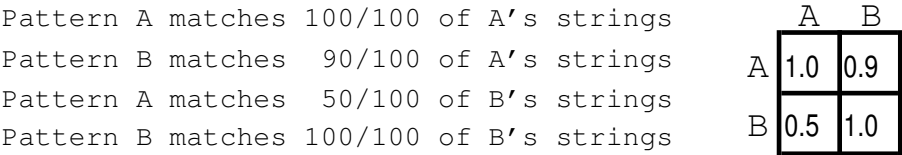
Figure 4.5   A similarity matrix created by counting strings matched

	A	B	C	D
A	1.0	0.0	0.9	0.0
B	0.2	1.0	0.8	0.7
C	0.6	0.8	1.0	0.2
D	0.0	0.6	0.1	1.0

→

	A	B	C	D
A	1.0			
B	0.1	1.0		
C	0.75	0.8	1.0	
D	0.0	0.65	0.15	1.0
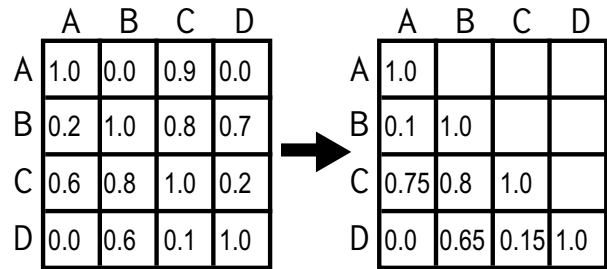
Figure 4.6   Creating a similarity graph from a similarity matrix

In this study, a regex similarity measurement technique was devised and used to group regex with similar behavior into clusters. The clusters representing the most projects were then in turn manually grouped into general categories of behavior.

4.2.1.1   Determining behavioral similarity

An ideal analysis of regex behavioral similarity would use subsumption or containment analysis. However, a tool that could facilitate such an analysis of the corpus could not be found. For this reason, a new technique using string matching was developed that can create a similarity score between two regexes with existing technology.

4.2.1.2   Building a similarity matrix

The similarity analysis used in this study clusters regular expressions by their behavioral similarity on matched strings. Consider two unspecified regexes `A` and `B`. Now consider a set of 100 strings that match `A`, named A100m. If `B` matches 90 strings in A100m, then If `B` is 90% similar to `A`. Similarly, consider a set of 100 strings that match `B`, named B100m. If `A` matches 50 strings in B100m, then If `A` is 50% similar to If `B`. Notice that by this definition, each regex is 100% similar to itself. Similarity scores are used to create a similarity matrix as shown in Figure 4.5.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix are averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 4.6. This facilitates clustering using the Markov Clustering (MCL) algorithm[5]. MCL was chosen because it offers a fast and tunable way to cluster items by similarity and it is particularly useful when the number of clusters is not known *a priori.*

In the implementation, strings are generated for each regex using Rex [Veanes et al. (2010)]. Rex generates matching strings by representing the regular expression as an automaton, and then passing that automation to a constraint solver that generates members for it[6]. If the regex matches a finite set of strings smaller than 400, Rex will produce a list of all possible strings. In this study, the goal is to generate 400 strings for each regex to balance the runtime of the similarity analysis with the precision of the similarity calculations.

For clustering, the similarity matrix is pruned to retain all similarity values greater than or equal to 0.75, setting the rest to zero, and then using MCL to actually find the clusters. This threshold (0.75) was selected based on recommendations in the MCL manual [van Dongen (2012)]. The impact of lowering the threshold would likely result in either the same number of more diverse clusters, or a larger number of clusters, but is unlikely to markedly change the largest clusters or their summaries. We also note that MCL can also be tuned using many parameters, including inflation and filtering out all but the top-k edges for each node [van Dongen (2012)]. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and k=83. The top 100 clusters are categorized by inspection into six categories of behavior.

The end result is clusters and categories of highly behaviorally similar regular expressions, though this approach has a tendency to over-approximate the similarity of two regexes. Similarity is measured based on a finite set of generated strings, but some regexes match an infinite set (e.g., `ab*c`), so measuring similarity based on the first 400 strings may lead to an artificially high similarity value. To mitigate this threat, a large number of generated strings was

---

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

used for each regex, but future work includes exploring other approaches to computing regex similarity.

4.2.2 Clustering implementation

4.2.2.1 Selecting regexes to include

Earlier runs of the study determining behavioral similarity attempted to include all regexes - only to find that because only the clusters representing the most projects could be included in further analysis, regexes belonging to only one project had little affect on the results. In light of this and also due to the computationally intensive nature of building the similarity matrix, regexes appearing in only one project were not included in the final behavioral analysis. So of the 13,597 regexes of the corpus, 10,015 (74%) regexes that were not found in multiple projects were not included. An additional 711 (5%) regexes were excluded that contain features not supported by Rex. The remaining **2,871** (21%) regexes were used in the similarity analysis technique described here.

The impact is that 923 (53%) projects were excluded from the data set for the similarity analysis. The remaining **730** (47%) projects touched by some cluster remained relevant to the analysis. Omitted features are indicated in Table 4.7 for Rex. Further details about similarity matrix creation are available in Appendix B.

4.2.2.2 Markov clustering details

The Markov Clustering Algorithm (MCL) is based on the principal that when taking a random walk within a graph, a random walker is more likely to stay within a cluster than to cross to another cluster [van Dongen (2012)]. A graph can be represented as a matrix of edge weights (the similarity matrix). One step of a random walk can be simulated by multiplying this matrix by itself (i.e., the matrix after one step has cells $c_{i,j}$ containing the sum of multiplying column i of M with row j of M.). This is known as *expansion* [van Dongen (2012)]. With MCL, the natural tendency of clusters to be emphasized using random walks is exacerbated by raising each matrix element to a power and then re-normalizing the matrix (so that random walks may

Table 4.8   An example cluster containing 12 regexes, with at least one regex present in 31 different projects. In this cluster, every regex requires ':'.

index	pattern	nProjects	index	pattern	nProjects
1	`\s*([^: ]*)\s*:(.*)`	9	7	`[:]`	6
2	`:+`	8	8	`([^:]+):(.*)`	6
3	`(:)`	8	9	`\s*:\s*`	4
4	`(:+)`	8	10	`\:`	2
5	`(:)(:*)`	8	11	`^([^:]*):[^:]*$`	2
6	`^([^:]*): *(.*)`	8	12	`^[^:]*:([^:]*)$`	2

continue). This is known as *inflation* [van Dongen (2012)]. Inflation effectively lowers smaller numbers more than large ones. Edges between nodes in a cluster tend to remain intact (they were not lowered so much during the random walk) while all other edges are effectively pruned. The algorithm works by alternating between n expansion steps followed by inflation to the power i. These steps are repeated until the matrix converges to a fixed point [van Dongen (2012)]. More detail about how MCL was tuned are available in Appendix B.

4.2.3   Categorization of behavioral clusters

From 2,871 distinct regexes, MCL clustering identified 186 clusters with 2 or more regexes, and 2,042 clusters of size 1. The average size of clusters larger than size one was 4.5. Each regex belongs to exactly one cluster. Five example clusters are available in Appendix B.

4.2.3.1   Representing a cluster with the shortest regex

Table 4.8 provides an example of a behavioral cluster containing 12 regexes. At least one regex from this cluster is present in 31 different projects. All regexes in this cluster share the literal `:` character. The smallest regex, `:+`, matches one or more colons.

Another regex from this cluster, `([^:]+):(.*)`, requires at least one non-colon character to occur before a colon character. The behavioral similarity score between these two regexes was below the minimum of 0.75 because Rex generated many strings for `:+` that start with

Table 4.9   Cluster categories and sizes, ordered by number of projects containing at least one pattern in the category.

Category	Clusters	Patterns	Projects	% Projects
Multi Matches	21	237	295	40%
Specific Char	17	103	184	25%
Anchored Patterns	20	85	141	19%
Two or More Chars	16	40	120	16%
Content of Parens	10	46	111	15%
Code Search	15	27	92	13%

one or more colons. However the overall similarity between the regex requiring a non-colon and other regexes in this cluster caused it to be clustered with this group.

The smallest regex in a cluster provides insight about key characteristic that all the regexes in the cluster have in common. A shorter regex will tend to have less extraneous behavior because it is specifying less behavior, yet, in order for the smallest regex to be clustered, it had to match most of the strings created by Rex from many other regexes within the cluster, and so one minor discovery about clusters of regexes is that the smallest regex is useful as a representative of the cluster. In this thesis, clusters will be represented using a shorthand composed of the shortest regex in the cluster followed by the number projects (and number of patterns in angle brackets). For Example, the cluster in Table 4.8 will be represented as `:+` (31 <12>).

The top 100 largest clusters based on the number of projects were manually sorted into 6 behavioral categories (determined by inspection). The largest cluster was left out, as it was composed of patterns that trivially matched almost any string, like `b*` and `^`. The remaining 99 clusters were all categorized. These clusters are briefly summarized in Table 4.9, showing the name of the category and the number of clusters it represents, patterns in those clusters, and projects. The most common category is *Multi Matches*, which contains clusters that have alternate behaviors (e.g., matching a comma or a semicolon, as in `,|;` (18 <5>). Each of the aforementioned 99 clusters was mapped to exactly one category.

4.2.3.2   Six categories of behaviorally similar clusters

Multiple Matching Alternatives   The patterns in these clusters match under a variety of conditions by using the CCC or OR feature. For example: `(\W)` (88 <36>) matches any alphanumeric character, `(\s)` (97 <33>) matches any whitespace character, `\d` (58 <23>) matches any numeric character, and `,|;` (18 <5>) matches a comma or semicolon. Most of these clusters are represented by patterns that use default character classes, as opposed to custom character classes. This provides further support for the survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which a majority of participants indicated they use the default classes more than custom. This category contains 21 clusters, each appearing in an average of 33 projects.

Specific Character Must Match   Each cluster in this category requires one specific character to match, for example: `\n\s*` (43 <16>) matches only if a newline is found, `:+` (31 <12>) matches only if a colon is found, `%` (22 <6>), matches only if a percent sign is found and `}` (14 <4>) matches only if a right curly brace is found. The commonality of this cluster category contrasts with the survey in (Section 4.3.2) in which participants reported to very rarely or never use regexes to check for a single character (Table 4.11). This category contains 17 clusters, each appearing in an average of 17.1 projects. These clusters have a combined total of 103 patterns, with at least one pattern present in 184 projects.

Anchored Patterns   Each of the clusters uses at least one endpoint anchor to require matches to be absolutely positioned, for example: `(\w+)$` (35 <8>) captures the word characters at the end of the input, `^\s` (16 <4>) matches a whitespace at the beginning of the input, and `^-?\d+$` (17 <2>) requires that the entire input is an (optionally negative) integer. These anchors are the only way in regexes to guarantee that a character does (or does not) appear at a particular location by specifying what is allowed. As an example, `^[-_A-Za-z0-9]+$` says that from beginning to end, only `[-_A-Za-z0-9]` characters are allowed, so it will fail to match if undesirable characters, such as '?', appear anywhere in the string. This category contains

20 clusters, each appearing in an average of 15.4 projects. These clusters have a combined total of 85 patterns, with at least one pattern present in 141 projects.

Two or More Characters in Sequence These clusters require several characters in a row to match some pattern, for example: `\d+\.\d+` (30 <7>) requires one or more digits followed by a period character, followed by one or more digits. The cluster ` ` (17 <4>) requires two spaces in a row, `([A-Z][a-z]+[A-Z][^ ]+)` (11 <2>), and `@[a-z]+` (9 <1>) requires the at symbol followed by two or more lowercase characters, as in a twitter handle. This category contains 16 clusters, each appearing in an average of 13 projects. These clusters have a combined total of 40 patterns, with at least one pattern present in 120 projects.

Content of Brackets and Parenthesis The clusters in this category center around finding a pair of characters that surround content, often also capturing that content. For example, `\(.*\)` (27 <7>) matches when content is surrounded by parentheses and `".*"` (26 <6>) matches when content is surrounded by double quotes. The cluster `<(.+)>` (23 <4>) matches and captures content surrounded by angled brackets, and `\[.*\]` (22 <7>) matches when content is surrounded by square brackets. This category contains 10 clusters, each appearing in an average of 18.4 projects. These clusters have a combined total of 46 patterns, with at least one pattern present in 111 projects.

Code Search and Variable Capturing These clusters show a recognizable effort to parse source code or URLs. For example, `^https?://` (13 <3>) matches a web address, and `(.+)=(.+)` (9 <2>) matches an assignment statement, capturing both the variable name and value. The cluster `\$\{([\w\-]+)\}` (11 <2>) matches an evaluated string interpolation and captures the code to evaluate. This category contains 15 clusters, each appearing in an average of 11.7 projects. These clusters have a combined total of 27 patterns, with at least one pattern present in 92 projects.

4.2.4   Discussion of cluster categories

When tool designers are considering what features to include, data about usage in practice is valuable. Behavioral similarity clustering helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in matching behavior. Many clusters are defined by the presence of particular tokens, such as the colon for the cluster in Table 4.8.

4.2.4.1   Implications

Supporting source code parsing   One of the six common cluster categories, *Code Search and Variable Capturing*, has a very specific purpose of parsing source code files. This shows a very specific and common use of regular expressions in practice. Further study is needed into how language designers can best support this practice.

Possible bracket parsing bugs   The *Content of Brackets and Parenthesis* cluster revolves around a parsing behavior that is generally frowned upon, since (most) regular expressions cannot accurately parse context-free languages. This implies the presence of potential code bugs, where programmers are using regexes to parse angle brackets in a way which may fail to behave as intended. More study is needed into how angle bracket parsing using regex effects source code.

Anchors have a strong behavioral effect   The identification of the *Anchored Patterns* category of clusters suggests that anchoring patterns is especially affecting on the behavior of regexes. Anchors were the only features that had enough of an effect on behavior that clusters formed based on their presence. Of the four regex analysis tools listed in Table 4.7, only two support the END anchor. This implies that tool designers should work to better support anchors.

Finding specific content using delimiters   Two categorical clusters, *Specific Characters Must Match* and *Two or More Characters in Sequence*, deal with identifying the presence of specific character(s). While multiple character matching subsumes single character matching,

the overarching theme is that these regexes are looking to validate strings based on the presence of very specific content, as would be done for many common activities listed in Table 4.11, such as, "Locating content within a file or files." More study is needed into what content is most frequently searched for, but this cluster analysis indicates that version numbers, twitter or user handles, hex values, decimal numbers, capitalized words, and particular combinations of whitespace, slashes and other delimiters were discernible targets.

4.2.4.2   Threats to validity

The following threats impact these results and conclusions:

Measures   The similarity measure between regexes used in the cluster algorithm is computed empirically rather than analytically, and the more Rex-generated strings used to compute the similarity measure, the more likely it is to be accurate. Our experiments used 400 strings to balance performance and precision, but a higher number could lead to more cohesive clusters. Additionally, regex patters that use any feature not supported by Rex were omitted from the cluster analysis. Last, the threshold of 0.75 was chosen based on the MCL recommendation, but it may not create optimal clusters.

Instrumentation   Regular expression patterns were clustered using strings generated by the Rex tool. We assume that the strings generated by Rex are reasonably diverse to help characterize the regex behavior. To mitigate this threat, Rex generated 400 strings per regex and we inspected strings randomly to ensure diversity.

4.3   Regex Usage By Surveyed Developers

When trying to assess how developers use regular expressions, input directly from developers is useful and relevant. The goal of this section is to answer the question, 'What preferences, behaviors and opinions do professional developers have about using regex?'. Professional software developers at Dwolla, a startup company focused on moving money, were asked about their usage and preferences when using regular expressions. The results of this survey indicate that developers use regular expressions most frequently in text editors and on the command line, have a strong preference for numbered over named backreferences, and about half of developers use online testing tools when composing regular expressions.

4.3.1   Survey design

The survey designed to understand the context of when and how programmers use regular expressions was designed and implemented using Google Forms containing 30 questions. All survey questions are in Appendix C. The survey is composed of self-qualifying questions, questions about contextual usage frequency (by task and by environment), feature usage, best practices (testing, and parsing HTML) and pain points. Participation was voluntary and participants were entered in a lottery for a $50 gift card.

4.3.1.1   Participants

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

4.3.2   Summary of survey results

The survey was sent to 22 employees at Dwolla, and 20 began a response with two eliminated by responding 'no' when asked if they were a professional software developer that had used regex in a work environment. The remaining 18 participants (82% response rate) have an average of nine years of programming experience ($\sigma = 4.28$).

4.3.2.1   Contextual usage frequency

To increase accuracy of recall, participants were first prompted to recall per-environment, and per-task before asking for the overall number of regexes composed per year. On average, survey participants report to compose 172 regexes per year ($\sigma = 250$) and compose regexes on average once per month, with 28% composing multiple regexes in a week and an additional 22% composing regexes once per week. That is, 50% of respondents uses regexes at least weekly.

Table 4.10   Survey results for number of regexes composed per year by technical environment

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

Usage frequency by technical environment   Table 4.10 summarizes how frequently participants compose regexes using each of several languages and technical environments (complete response data is available in Appendix C.) Six (33%) of the survey participants report to compose regexes using general purpose programming languages (e.g., Java, C, C#) 1-5 times per year and five (28%) do this 6-10 times per year. For command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year. Yet, regexes were rarely used in query languages like SQL. Upon further investigation, it turns out the surveyed developers were not on teams that dealt heavily with a database.

Table 4.11   Survey results for regex usage frequencies for tasks, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

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

Usage frequency by task category   Table 4.11 summarizes how frequently, on average, the participants use regexes for various tasks. Participants answered questions using a 6-point likert scale including very frequently (6), frequently (5), occasionally (4), rarely (3), very rarely (2), and never (1). Complete response data is available in Appendix C. Averaging across participants, among the most common usages are capturing parts of a string and locating content within a file, with both occurring somewhere between occasionally and frequently.

4.3.2.2   Feature usage questions

Rarely used OPT and DBB features   When asked if they have ever used the OPT feature ((?i)), 78% (14) said that they had never used it, with the rest saying they had. However the reverse is true for the DBB feature, with 78% (14) saying they *had* used it before, and the rest saying they had not.

Use of the WRD default   A special investigation was made into the WRD default after a trend in the corpus was observed where, with some frequency, custom character classes are made that slightly modify the WRD default. For example, the regex `[\w-]` might be used to represent characters allowed in a username. Participants were asked what they use the WRD

Table 4.12   Survey results for regex usage frequencies, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

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

Table 4.13   Responses to survey Q18: If you know it won't affect your use case would you prefer to use '.*' or '.+' ?

Always KLE	Mostly KLE	Mostly ADD	Always ADD
7	5	4	2

default character class for, but the intention of the question may not have been clear enough, because 50% (9) participants said they don't know or don't use it, and 44% (8) participants said they use it to match WRD characters. However, one participant said 'often we pair it with other characters we know are found in traditional writing such as punctuation marks.'.

Features poorly supported by analysis tools   Participants were asked about how frequently they used five feature groups poorly supported by analysis tools. Their responses are shown in Table 4.12, indicating that lazy repetition and look-ahead features are rarely used and capture groups and endpoint anchors are occasionally to frequently used. Complete response data is available in Appendix C.

4.3.2.3   Refactoring preference questions

Pairing ANY with KLE or ADD   Participants were asked if, given a guarantee that it would not affect their use case, they would prefer to use ANY with KLE (*) or with ADD (+). Participants generally preferred to use KLE, with 39% (7) saying 'always use `.*`', 28% (5)

Table 4.14   Survey results for preferences between custom character and default character
classes

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

saying 'use `.*` more than `.+` ', 22% (4) saying 'use `.+` more than `.*` ', and 11% (2) saying
'always use `.+` '. These results are displayed in Table 4.13. Semantically, there is a difference
between these two, which triggered a rich, skeptical response in the section asking them to
explain their preference. Aside from this understandable skepticism, those who preferred ADD
were referring to the expected content, and those who preferred KLE said that they are 'used
to using' KLE or 'need ∗ more often'.

CCC vs defaults   Semantically equivalent regexes can be created using the CCC (`[...]`)
feature, or using default character classes like DEC (`\d`) (e.g, `[0-9]` ≡ `\d`). Survey partic-
ipants were asked if they use only CCC, use CCC more than default, use both equally, use
default more than CCC or use only default. Results for this question are shown in Table 4.14,
with 67% (12) indicating that they use defaults the most. Participants who favored CCC indi-
cated that "it is more explicit," whereas the participants who favored default character classes
said, "it is less verbose" and "I like using built-in code."

Named vs Numbered Backreferences   Participants were asked whether, assuming a
backreference was needed, they would 'always use numbered backreferences', 'it depends', or
'always use named backreferences'. No participants said that they would always use named
backrefernces, with 33% (6) participants saying 'it depends', and 67% (12) saying they would
always use numbered backreferences. For the three participants who responded to why 'it
depends', two said that the more captures they need, the more likely they are to use named

backreferences. The third response seemed to indicate that they have never needed to use backreferences.

	always	v. freq	freq.	occ.	rarely	v. rarely	never
test code	4	7	5	1	0	0	1
test regex	3	4	5	5	1	1	0

Table 4.15   Q5: Please describe how often you compose regex for a particular problem type.

4.3.2.4   Best practices and pain points

Code testing vs regex testing   Participants answered these questions using a 7-point likert scale using always (7) as a seventh point in addition to the the six points outlined in Section 4.3.2.1. Developers were asked how frequently they test their code, with 89% (16) indicating that they test their code at least frequently. Developers were also asked how frequently they test their regexes, with 89% (16) indicating that they test their code at least occasionally. These results are summarized in Table 4.3.2.3. Comparing the testing frequency between regex and other code for each participant, 39% (7) test code more frequently than regex, 44% (8) test both with the same frequency, and 17% (3) test regex more frequently than code. Prior work suggests that regexes are hard understand, causing tens of thousands of reported bugs per year [Spishak et al. (2012)]. This finding is reinforced by the trend in participant responses of testing code more than regex.

Regex testing tools   When asking if they use testing tools, 50% (9) indicated that they use some tool, with 33% (6) using an online tool like regex101.com where a regex and input string are entered, and the input string is highlighted according to what is matched. The remaining 17% (3) developers who used testing tools used Scalacheck, Regexlib and IDE plugins. Of the 9 developers who did not use testing tools, 17% (3) indicated that they write their own tests for their regexes, and 33% (6) answered 'none'.

Parsing HTML   Because parsing a markup language like HTML using regular expressions is a common mistake[7], (typically an HTML parser is a better choice) participants were

---

[7] http://stackoverflow.com/questions/1732348

asked if they had ever tried to parse HTML or XML. Participants were also asked if they would rather use a custom parser or a regex, and why. Trying to parse HTML with regular expressions is a mistake, because regular expressions do not handle the balancing of opening and closing tags well. Such balanced tags are an expression of a context free language, and regular expressions (mostly) express regular languages (are not as expressive). On the other hand there are sufficient HTML parsers available in every major language that do not suffer from this problem at all, so programmers are generally recommended to use a parser to parse HTML, and not regexes. Only 28% (5) of the participants admitted to using regexes on HTML. When asked if they would 'use only custom parser' or 'use only regex', or 'it depends', 22% (4) said they would favor the parser, and only 11% (2)said they would favor regex. The remaining 67% (12) said 'it depends'. Of those who said it depends, there was much agreement that parsers are more powerful, better at handling complexity, and provide more readable code. Regexes were mentioned positively for handling less structured inputs, form validation and 'one-off' programs.

Pain points and free responses Participants were asked an open-ended question about problems with regular expressions: 'What pain points have you encountered with regular expressions?'. Three main categories of response were observed. The most common, "hard to compose," was represented in 61% (11) responses. Next, 39% (7) developers responded that regexes are "hard to read" and 17% (3) indicated difficulties with "inconsistency across implementations," which manifest when using regexes in multiple languages. These responses do not sum to 18 as three developers provided multiple parts in their answers. Complete response data is available in Appendix C. Participants were given a chance to add anything else that had not been asked about. Many comments conveyed a sense of mixed feelings - that regex are useful but require caution.

4.3.3 Comparing ephemeral and persistent users

Some of the main applications for regexes, such as searching text files and system administration (Section 3.2), do not leave a persistent artifact, like a text file, behind. Since most of

Table 4.16 Survey results for regex usage frequencies, comparing persistent and ephemeral users

Group	Code	Ephemeral Users	Persistent Users	Difference
(neg) look-ahead/behind	(LKA, NLKA, LKB, NLKB)	2.2	3.2	1.0
lazy repetition	(LZY)	2.8	3	0.2
endpoint anchors	(STR, END)	4.4	4.4	0
capture groups	(CG)	4.2	4.2	0
word boundaries	(WNW)	3.5	3.4	-0.1

the analysis in this work is based on persistent regexes (i.e., mined from GitHub repositories), this section analyzes differences between these two types of users. *Ephemeral users* are those who primarily use regexes that are used once and then forgotten, and *persistent users* are those who primarily use regexes that are maintained as an artifact. In question four of the survey, participants were asked to recall the number of regexes composed per year by technical environment. Only 27% (5) of participants wrote regular expressions that persist (general purpose, scripting, etc.) more frequently than in a text editor or command line tool (where they will not persist).

Persistent user characteristics   The five persistent users have an average of 12.4 years of experience. This contrasts with an average of 7.7 years of experience for ephemeral users. Considering usage frequency, 60% (3) of the five persistent users indicate using regex weekly, vs 46% (6) of the 13 ephemeral users.

Persistent users use more advanced features   Of the five feature groups analyzed, the only significant difference in usage is for the lookaround group (LKA, LKB, NLKA, NLKB) for which persistent users indicated an average between rarely and occasionally, compared to ephemeral users who averaged between very rarely and rarely. The differences for all five groups are shown in Table 4.16. In the case of the most rarely-used feature asked about, the OPT feature, three of the four participants who have ever used the OPT feature are persistent users.

Table 4.17   Results of subtracting the average task frequency of ephemeral users from the
average task frequency of persistent users, ordered by difference

Task	Persistence Freq.	Ephemeral Freq.	Difference
Counting substrings that match a pattern	3	1.7	1.2
Parsing user input	3.6	2.7	0.9
Capturing parts of strings	3.8	3.1	0.7
Parsing generated text	2.4	1.9	0.5
Locating content within a file or files	3.6	3.2	0.4
Filtering collections (lists, tables, etc.)	2.2	1.9	0.3
Counting lines that match a pattern	1.8	2.1	-0.3

Persistent users perform different tasks   The five participants who write persistent regexes more often also answered the task frequency questions differently. Table 4.17 describes the tasks more frequently performed by persistent users than by ephemeral users. Most notably, persistent users are counting strings and parsing generated text with greater frequency than 'rarely'. This makes sense because these are use cases more suited for a complex software program than a quick search.

4.3.4   Discussion of survey results

The fact that all the surveyed developers compose regexes, and half of the developers use tools to test their regexes indicates the importance of tool development for regex. Developers complain about regexes being hard to read and hard to write, and express a respect for the power of regexes but also a hesitancy or caution concerning their use. This supports the need for research into regular expressions to improve the state-of-the-art, especially to help developers leverage the power of regular expressions with more confidence. Participants reported testing code more frequently than testing regexes (Section 4.3.2.4), and though some online tools exist, guaranteeing coverage of all corner cases is challenging. More research is needed into what kind of support is required to help developers use regular expressions with complete confidence.

Common uses of regexes include locating content within a file, capturing parts of strings, and parsing user input. Although ephemeral users are more common than persistent users, persistent users tend to use regexes more frequently than ephemeral users in a variety of tasks types, especially in counting substrings, parsing user input, capturing strings and parsing generated text. This implies that improving the state-of-the-art for different classes of regular expression users will mean focusing on different goals.

In terms of refactoring implications, developers communicated a preference for using default character classes when possible, using numbered capture groups over named capture groups, and choosing KLE over ADD if all else is equal. refer to refactorings?

4.3.4.1 Threats to validity

The greatest threat to validity is that this is a small sample set. Future study is needed, using a larger sample set to obtain more statistically sound results. Also, the population of developers at Dwolla may be more homogeneous than the general population of programmers - more study is also needed to gather the type of data gathered in this study from a diverse population of programmers.

Another threat to validity is that survey participants may tend to say what they believe they are supposed to say. This threat is mitigated by the disclaimer given at the beginning of the survey to 'answer as accurately as possible, not guessing what is the desired answer and providing that'.

4.4 Regex Refactorings Based On Community Standards

4.4.1 Defining five equivalence classes

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. This topic will be explored by answering the research question,
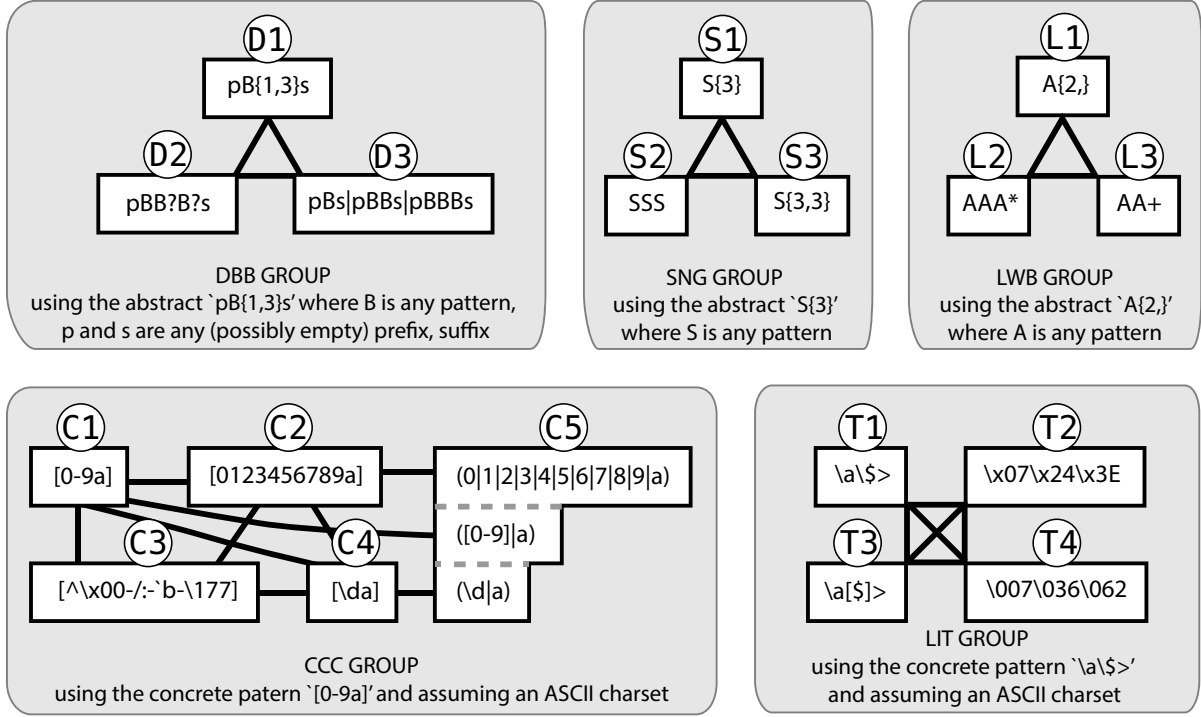
Figure 4.7   Equivalence classes with various representations of semantically equivalent representations within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

'Within five equivalence classes, what representations are most frequently observed?' 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. This investigation focuses on preferences in terms of community standards, established by which representation appears most frequently in source code.

Figure 4.7 displays five equivalence classes in grey boxes. This work will often use the term *group* as shorthand for 'equivalence class'. These equivalence classes were chosen by considering what alternatives are possible between the most commonly used features from Section 4.1.4. Three classes focus on representations of repetition (SNG, DBB and LWB), CCC focuses on representations of character classes, and LIT focuses on how characters are represented. Additional classes of behaviorally identical regexes are discussed in Section 6.1.1.

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.

Figure 4.7 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. The lower bound repetition threshold of 2 provides a useful illustration, but is not meant to describe a requirement for the equivalence class. The analysis performed for this study uses a lower bound of 1. Further study is needed on how specific lower bounds, and other variables, may affect preferences among nodes.

4.4.1.1   Summary of node membership criteria

Criteria for node membership is summarized in this section. For a more detailed description, see Appendix D

C1: Any regex using the RNG feature in a CCC like `[a-f]` belongs to C1.

C2: Any regex that contains at least one CCC without any RNG or defaults like `[aeiou]` belongs to the C2 node.

C3: Any regex using the NCCC feature like `[^x]` belongs to the C3 node..

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) like `(a|\d|x)` belongs to the C5 node.

D1: Any regex that uses the DBB feature, such as `pB{1,3}s` , belongs to the D1 node.

D2: Any regex that uses the QST feature like `xyz?` belongs to D2.

D3: Any regex that uses OR to express repetition with different upper and lower boundaries like `pBs|pBBs|pBBBs` belongs to D3..

T1: Patterns using ordinary literal characters, and not belonging to other nodes in the LIT group, like `a` belong to the T1 node.

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, like `[x][y][z]` belongs to T3.

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

L1: Any regex using the LWB feature like `A{3,}` belongs to the L1 node.

L2: Any regex using the KLE feature like `X*` belongs to the L2 node.

L3: Any regex using the ADD feature like `T+` belongs to the L3 node.

S1: Any regex using the SNG feature like `S{3}` belongs to the S1 node.

S2: Any regex that is explicitly repeated two or more times, and could use repetition operators, like `(ab)(ab)` or `ggg` belongs to the S2 node.

S3: Any regex with a double-bound in which the lower and upper bounds are same belongs, like `Z{3,3}` to S3.

4.4.1.2   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]`.

4.4.2   Counting representations in nodes

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 4.4.1. 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. The final outcome of this node counting process is summarized in Table 4.18.

The source code used to perform the node counting process is available on GitHub[8].

4.4.3 Node counting results

For each node, Table 4.18 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 4.7. 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 the corpus. These appear in 810 projects, representing 52.5%. Regexes belonging to D1 appear

---

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

Table 4.18   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%

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.

4.4.4   Discussion of refactorings

Using the count of regexes in each node provided in Table 4.18, 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.

4.4.4.1   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, most regexes in C2 do not express ranges of characters, but instead express
non-continuous sets. The following regexes (or regex fragments) extracted from the cor-
pus illustrate this point: `[?/:|]+` , `coding[:=]` , `([\\"]|[^\ -~])` , `^[012TF\*]{9}$` ,
`\?|[-+]?[.\w]+$` . None of these regexes can be refactored out of C2.

Refactoring out of C3 is generally awkward On inspection, most regexes in C3
seem to be negating just one or two characters like `[^:]*:` and `^([^/:]+):` , and in these
cases refactoring out of C3 would result in a strange-looking regex. 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]+):` . Based on logical reasoning about how
to express the negation of a character set without using NCCC, the most notable candidate
for refactorings going out of C3 is from C3 to C4, because many NCCC simply represent the
negated version of some default character class. However, according to community standards
the preferred representation may be in C3, not C4. 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.

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 4.18. 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. These

are probably not accidental omissions, but likely stem from a familiarity with the problem space being dealt with by the regex (e.g., newlines are not expected, so they are not included in the CCC).

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 `^(?:!|&|\\*)$` to `^(?:[!&*])$`, respectively.

4.4.4.2   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 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.

4.4.4.3 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.

4.4.4.4 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.

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. Considering how common english words with double letters are, including regexes with double letter words as part of S2, as is done in this analysis, over-estimates the real number of semantically equivalent regexes for which a refactoring is reasonable. Transforming regexes using S1 on ordinary characters, like `lit{2}le fo{2}t` to double letters like `little foot` can be recommended from the data. However, for other types of refactoring, a side-effect of including double letters in S2 is an over-estimation of the community support for refactoring into S2 for regexes that are not dealing with repeating letters in words. 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. One suggestion is to create a

separate equivalence class dealing with S2-type repetition of ordinary characters, dealing with other elements separately.

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 large segment of S2 based on double letters in words, 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.

4.4.4.5   Threats to validity

The technique of determining node counts includes manual verification, so it 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 were very simple, and may not have been sophisticated enough to consider all the nuances of real usage. For example, additional rules could have been applied to split the SNG equivalence class into one class dealing with SNG-type repetition applied exclusively to ordinary characters, and one class not dealing with ordinary characters. Hypothetically, opposite refactorings could be preferred in these two groups. By merging them, the refactoring preferences would interfere and no preference would be detected. As the first work on regular expression refactoring, this outcome is not unexpected. Other equivalence classes are considered in Section 6.1.1.

Since 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.

4.5   Regex Refactorings Based On Comprehension

The goal of this study is to answer the question, 'Within five equivalence classes, what representations are more comprehensible?' by presenting programmers with one of several representations of semantically equivalent regexes and asking comprehension questions. By comparing the understandability of semantically equivalent regexes that have different representations, it is possible to infer which representations are more desirable. 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 4.7.

Table 4.19   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

4.5.0.6   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 match*, *unsure*, or blank. An example from the study is shown in Figure 4.8. The use of the term 'matches' in this chapter is consistent with the meaning described in Section 2.2.2 - 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.

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 4.19, and the

Subtask 7. Regex Pattern: `'((q4f)?ab)'`

7.A	`'qfa4'`	○ matches	● not a match	○ unsure
7.B	`'fq4f'`	○ matches	● not a match	○ unsure
7.C	`'zlmab'`	○ matches	○ not a match	● unsure
7.D	`'ab'`	○ matches	○ not a match	● unsure
7.E	`'xyzq4fab'`	● matches	○ not a match	○ unsure

7.F Compose your own string that contains a match: `4q4fab`

Figure 4.8   Example of one HIT Question

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. An investigation into the potential meaning of unsure results is in Appendix E. A matching problem with all blank or unsure responses is 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 the 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.

4.5.0.7 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.

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. The qualification is available in Appendix E.

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. The last four metagroups contain two sets of three equivalent regexes. A list of the specific regexes selected for these metagroups is available in Appendix E.

M1 S1 vs S2

M2 C1 vs C4, focusing on DEC

M3 C1 vs C4, focusing on WRD

M4 C4 vs (C3 or C2)

M5 L2 vs L3

M6 T1 vs T3

M7 D1 vs D2 vs D3

M8 C1 vs C2 vs C5

M9 C2/T1 vs C5/T1 vs C2/T4

M10 C1/T2 vs C1/T4 vs C2/T1

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), seven total comparisons had to be dropped due to design flaws. For six of the comparisons, the regexes performed transformations from multiple equivalence classes, making it impossible to tell which edge to attribute the results to. 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. The seventh pair: `\..*` and `\.+` between L2 and L3, had to be dropped because these two regexes are not equivalent. The first regex was meant to be `\.\.*`. Data gathered for all seven of these flawed pairings was ignored.

An example of a correct 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 4.7.

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. A list of the specific strings created is available in Appendix E.

Once all the regexes and matching strings were collected, tasks for the MTurk participants were created 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. The randomly selected regexes and shuffled matchings strings populate fields in an HTML template (Appendix E). This template also includes a request for participants to compose a string that

matches each of the 10 regexes. Each populated template created one HIT. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs.

Worker statistics 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. All worker composition answers were inspected to make sure that the composition answer was composed in a good-faith effort to match the regex before accepting the HIT (the field was not empty, and the content seemed like it had required some thought). One worker 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.

4.5.1 Population characteristics

In total, there were 180 participants in the study. A majority were male (83%) with an average age of 31. Most had at least an Associates degree (72%) and most were at least somewhat familiar with regexes prior to the study (87%). On average, participants compose 67 regexes per year with a range of 0 to 1000. Participants read more regexes than they write with an average of 116 and a range from 0 to 2000. Figure 4.20 summarizes the self-reported participant characteristics from the qualification survey.

4.5.2 Matching and composition comprehension results

For each of the 180 HITs, a matching and composition score was computed for each of the 10 regexes, using the metrics described in Section 4.5.0.6. 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

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

1.

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

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%

3.

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%

4.

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?**
   $\mu = 116$, $\sigma = 275$

Table 4.20   Participant Profiles, $n = 180$

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 35 pairwise comparisons.

The regexes `((q4f)?ab)` and `(q4fab|ab)`, represent a transformation between D2 and D3. The former have an average matching score of 79% and the latter have an average matching score of 85%. The average composition score for the former is 83% and 97% for the latter. Thus, the community found `(q4fab|ab)` from D3 more understandable. The other pairwise comparison performed between D2 and D3 uses the pair `(deedo(do)?)` and `(deedo|deedodo)`. Considering both of these regex pairs, the *overall matching score* for the regexes belonging to D2 is 78% and the *overall matching score* for D3 is 87%. The *overall composition score* for D2 is 88%, with 97% for D3. Thus, the community found D3 to be more understandable than D2, from the perspective of both understandability metrics, suggesting a refactoring from D2 to D3. These measurements comparing overall representation understandability are presented in summary in Table 4.21, with this specific example appearing in the E2 row. The pairs of

Table 4.21   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

regexes used to determine the values for each edge are listed in Appendix E. 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.

In Table 4.21, a thin horizontal line separates the top four edges from the bottom 10. In these four edges, there is a statistically significant difference between the representations for at least one of the metrics considering $\alpha = 0.10$. These represent the strongest evidence for suggesting the directions of refactoring based on the understandability metrics defined in this study. Specifically, $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, $\overrightarrow{C2C5}$ and $\overrightarrow{C2C4}$ are likely to improve understandability. The specific nodes, regexes, matching scores and compositions scores that led to these refactoring suggestions are shown in Table 4.22

Table 4.22   Equivalent regexes with a significant difference in understandability, $\alpha = 0.1$

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%
C2	`([:;])`	81%	87%		C5	`(:|;)`	94%	100%
C2	`no[wxyz]5`	87%	90%	$\overrightarrow{C2C5}$	C5	`no(w|x|y|z)5`	94%	97%
C2	`([}{])`	79%	87%		C5	`(\{|\})`	70%	93%
C2	`tri[abcdef]3`	93%	90%		C5	`tri(a|b|c|d|e|f)3`	86%	90%
C2	`[\t\r\f\n ]`	83%	60%	$\overrightarrow{C2C4}$	C4	`[\s]`	92%	67%

In addition to the edges with a significant difference in understandability, edges E5 through E8 exhibit noteworthy trends in understandability. The regexes and matching scores for these nodes are shown in Table 4.23. For edges E7 and E8, the potential refactorings $\overrightarrow{C2C1}$ and $\overrightarrow{T4T2}$ are also suggested by the community standards determined in Section 4.4.3. E5 is also notable because in *both matching and composition*, all scores consistently indicate a small preference for L3 over L2 (5% in overall matching and 3% in overall composing). More work is needed to determine if a significant difference in understandability exists between these nodes.

Table 4.23   Additional equivalent regexes for which some preference in understandability is suggested

node	regex	match	compose	ref.	node	regex	match	compose
L2	`zaa*`	87%	97%	$\overrightarrow{L2L3}$	L3	`za+`	91%	100%
L2	`RR*`	86%	97%		L3	`R+`	92%	100%
D2	`((q4f)?ab)`	79%	83%	$\overrightarrow{D2D1}$	D1	`((q4f){0,1}ab)`	83%	97%
D2	`(deedo(do)?)`	77%	93%		D1	`(dee(do){1,2})`	85%	90%
C2	`tri[abcdef]3`	93%	90%	$\overrightarrow{C2C1}$	C1	`tri[a-f]3`	94%	97%
C2	`no[wxyz]5`	87%	90%		C1	`no[w-z]5`	93%	90%
T4	`xyz[\0133-\0140]`	71%	33%	$\overrightarrow{T4T2}$	T2	`xyz[\x5b-\x5f]`	79%	60%
T4	`t[\072-\073]+p`	90%	70%		T2	`t[\x3a-\x3b]+p`	89%	70%

4.5.3 Discussion of comprehension results

4.5.3.1 Implications

Four statistically significant refactorings for understandability: $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, $\overrightarrow{C2C5}$ and $\overrightarrow{C2C4}$ were identified by the results presented in Table 4.21. A detailed view of the results for these refactorings is presented in Table 4.22. The refactoring $\overrightarrow{T4T1}$ is also indicated by the analysis of community support in Section 4.4.3.

Four additional trends in understandability results were found that are not statistically significant with $\alpha = 0.1$, but show potential for the refactorings, $\overrightarrow{L2L3}$, $\overrightarrow{D2D3}$, $\overrightarrow{C2C1}$ and $\overrightarrow{T4T2}$. Two of these, $\overrightarrow{D2D3}$ and $\overrightarrow{T4T2}$, are refactorings also indicated by the analysis of community support in Section 4.4.3.

T4 to T4 is always recommended 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.

D2 to D3 is recommended for small repetitions 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.

Refactoring out of C2 needs more study Refactorings out of C2 to (C1, C4 and C5) all displayed a strong trend for improving understandability. More study is needed into determining which node to choose when refactoring out of C2.

4.5.3.2   Threats to validity

Mechanical Turk may not be an ideal source for regex comprehension study participants. This threat was mitigated by requiring workers to pass a pre-qualification test, and by checking the composed regexes for potential validity before accepting any HIT, as mentioned in Section 4.5.0.7.

Poor understandability could be due to an overly long and complex regex, instead of the representation being tested. This risk is mitigated by composing regexes of approximately equal difficulty, as much as possible, from the perspective of the author.

Design flaws have reduced the coverage of equivalence classes, so that not all refactoring possibilities are fully explored. Several improvements to the experiment design are possible and with the benefit of experiences gained during the course of this study, a superior experiment could be executed. However, with the rigorous treatment of excluding all data that was obtained under misconceptions, and making due with a partial result, this experiment retains valid results.

CHAPTER 5.   DISCUSSION

organize final discussion

5.1   Review Of Implications

The most frequently performed task according to Table 4.11 is 'locating content within a file or files'. This result agrees with the idea that regexes are used more often in text editors and command line tools than in general purpose languages, since locating content is often done within a text editor.

The eight most common features are found in over 50% of the projects. Shown in Table 4.3, the STR and END features are present in over half of the scanned projects containing utilizations. In our survey, over half (56%) of the respondents answered that they use endpoint anchors frequently or very frequently, and none of them claimed to never use them.

The LZY feature is present in over 36% of scanned projects with utilizations, and yet was not supported by two of the four major regex projects we explored, brics and RE2. In our developer survey, 11% (2) of participants use this feature frequently and 6 (33%) use it occasionally, showing a modest impact on potential users.

5.2   Additional Implications

The features of Rex in particular are of importance in this thesis, as Rex was used, indirectly, to determine a similarity score between regexes, as described in Section 4.2.1. The lack of support for various features reduced the number of features that could be included by .

contrast developer preferences with refactorings from sections

Backslash explosion - how does it affect readability?

KLE refactorings - suggested by readability with p=0.06,

When asked if they have ever used the OPT feature (`(?i)`), 78% (14) said that they had never used it, with the rest saying they had. This provides some context to Table 4.3 - if a feature is used only 9.4% of projects, then many programmers may have never even used it. But DBB was at 14.5% and in this case, 78% *had* used it. So there may be a cutoff around 10%...

It depends, but IMHO the capture group really shines in programming-language use, because captured content can be put into a variable and used later.

Simple matching that requires the whole string to match seems less useful - unless we are validating user input.

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.

5.2.1   Can't see the forest for the antecdotal evidence

CHAPTER 6.   FUTURE WORK

6.1   Refactoring Regexes

Much work remains to be done in the new field of regex refactoring. The techniques described in this study for identifying refactorings based on community standards and understandability can be applied to many other data sets, and can be refined or extended to include new strategies for identifying preferred representations.

6.1.1   Equivalence models

The refactoring studies in this work used 5 equivalence classes, each with 3 to 5 nodes to reason about possible refactorings. These equivalence classes are very inclusive of regexes with very different behavior, and are defined largely by the features used by a regex. This is not the only way to reason about refactoring regexes. Other possible approaches are discussed in this section.

Other known feature-based equivalences   Due to the functional variety and significant number of features to consider, this work does not provide a list of all possible feature-based refactoring groups. However the following 5 additional equivalence classes are examples of other possible groups:

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

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

Multi line option `(?i)[a-z]` $\equiv$ `[A-Za-z]`

Backreferences `(X)q\1` $\equiv$ `(?P<name>X)q(?P:<name>)`

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

Community-based equivalence classes   More narrowly defined equivalence models, specific to particular behaviors of the most frequently observed regexes in a community, would be inherently impactful. For example a node could require the presence of a very specific CCC like `[a-zA-Z0-9_-]` (which can alternatively be represented as `[\w-]`) that is frequently observed in a particular community. The results of a preference evaluation would necessarily be impactful because the node was designed to apply to that community.

Combining categorization, clustering and formal tools   Using formal reasoning tools like Microsoft's Automata.Z3, *every* regex (using supported features) can belong to a single cluster of exactly equivalent regexes. In this case, an entire body of regexes can be covered by an understandability study. Computational limits (when determining equivalence) may present a challenge in a thorough analysis of this type, but using the categories of regex usage defined in Section 4.2.3.2, the number of pairwise comparisons can be greatly reduced to only comparisons within a category. Future work is needed to determine the feasibility of this approach.

Approximating equivalence   In existing refactoring work, code after a refactoring cannot behave differently than it did before it was refactored. However, it is likely that for many common use cases, like parsing dates or emails, two non-equivalent regexes that have nearly identical behavior, where the differences never apply in practice, could be considered approximately equivalent. With appropriate levels of test coverage, risks of a bad transformation could be mitigated. Future work is needed into how approximate equivalences could be useful in the genre of regex refactoring.

6.1.2   Identifying Preferred Representations

Refactoring for performance   The representation of regexes may have a strong impact on the runtime performance of a chosen regex engine. Prior work has sought to expedite the processing of regexes over large bodies of text [Baeza-Yates and Gonnet (1996)]. Refactoring

regexes for performance would complement those efforts. Further study is needed to determine which representations are most efficient in general, and for each engine specifically.

Refactoring to prevent catastrophic backtracking Pathological regexes like `(a+)*b` [1] represent an avenue for attack on shared systems [Kirrage et al. (2013)]. This regex is not describing a complicated set of strings, but suffers from *catastrophic backtracking* because of a common engine implementation choice. On the author's machine in a Python shell, this regex takes about 30 seconds for an input of 28 'a's. Assuming the time doubles for every 'a' added (anecdotally true on author's machine), an input of 67 'a's will take over a million years to determine that no match can be found. This regex can be refactored to `a*b`, which matches the same set of strings, and completes both searches very quickly. The ability to cause catastrophic backtracking provides malicious users an opportunity to perform an algorithmic complexity attack, crippling shared machines that allow users to execute arbitrary regular expressions. So this refactoring, applied before regexes are executed, renders bad actors harmless, providing non-malicious users with greater freedoms on shared systems.

Refactoring for compatibility As discussed throughout this thesis, different language variants have different feature sets. In the case of transforming code in one language to code in another language [Nguyen et al. (2015)], regexes also must be refactored so that regexes in the transformed code use only features supported by the target language, and must maintain expected behavior. Besides transforming code, an organization may want to enforce standards for compatibility with a regex analysis tool like Z3, HAMPI, BRICS or REX.

Refactoring for obfuscation In certain applications where the actual behavior of a regex needs to be hidden from malicious users (e.g., a regex that obtains a password hash from a block of text), a *reverse* understandability refactoring may be appropriate. In this scenario, a regex is composed to perform the desired function, and then repeatedly refactored so as to become less understandable until the regex is very difficult to understand (and therefore

---

[1] http://www.rexegg.com/regex-explosive-quantifiers.html

difficult to exploit). Bad actors who obtain the sensitive source code still cannot understand what it does, and therefore cannot use it as a vulnerability.

6.1.3 Applications for regex refactoring

Regex migration libraries This work identified opportunities to improve the understandability of regexes in existing code bases by looking for some of the less understandable regex representations, which can be thought of as antipatterns, and refactoring to the more common or understandable representations. Building migration libraries to accomplish these refactorings and other yet-undiscovered regex refactorings, is a promising direction of future work to ease the manual burden of this process, similar in spirit to prior work on class library migration [Balaban et al. (2005)].

Regex 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.

6.2 Semantic Search

Given a large set of regexes, and a set of strings specified to either not-match or match, the problem of semantic search is to find the regexes that meet the matching and non-matching specifications. This would be useful if, for example, a programmer wants to find a regex to perform some complex, already-solved task and would rather re-use an existing correct regex, than re-invent the wheel. The brute-force approach will work, checking every regex against every string to determine if a match exists, and returning the regexes that meet the specification. However, in order to be viable the technique must scale to accommodate a very large number of regexes, or the searched set would be unlikely to contain a useful regex. Also, for a task complex enough to require semantic search, a very large number of string matching specifications should be expected.

Ideally, a relatively small number of regexes could be used to navigate the set of regexes, finding the regexes that do match the specifications without performing very many evaluations.

6.2.1   Finding a filter set

One technique devised by the author (but never implemented) for minimizing the number of regexes that need to be evaluated to solve semantic search is described here as 'finding a filter set'. This is left as an opportunity for future work.

The intuition is best demonstrated with an example. The regex `::` consists of only two ':' characters, and the five regexes `::\(.*\)`, `\s*::\s*`, `e*d::`, `std::regex` and `d:{2,6}` also all contain two ':' characters, but describe a more limited set of strings. Now the first semantic search is performed by user A: find a regex that matches `"abc"`.

Let `'::'` be the top filter for the other five regexes, and `e*d::` be a second layer of filter for the last two. The top filter is checked first, and it does not match the string, so no further evaluations are needed (no result is returned).

Now user B performs another search: match `" :: "`. This string is matched by the top filter, so the next three must be evaluated. The regex `::\(.*\)` requires a parenthesis character so it does not match. The regex `\s*::\s*` does match the string, and the second layer filter, `e*d::` does not match the space characters on either side. Because the second layer filter does not match, the last two regexes do not need to be evaluated.

Formally, `::` describes a set of strings that *subsume*, or *contain* the sets of strings described by the other five regexes, so it can be used as a filter. However, the regex `:` also subsumes these five regexes, but is not as good of a filter, because it is narrower, and therefore blocks fewer unnecessary searches. Furthermore, the regex `.` has the same width as `:`, but blocks even fewer unnecessary searches.

Finding a filter set on one level can be formally described as follows:

Consider a universe R of regexes, and strings s from the set S over alphabet Σ.

Each $r \in R$ has matching function $r.m(s)$ returning true if r matches s (using the definition of matching described in Section 2.2.2), false otherwise.

Now find n *effective* filters $f_i \in U$, $0 \leq i < n$, where each filter maps to a minimally-overlapping, roughly equal-sized region of U: U_i.

A filter f_i is *effective* if $\forall s \in S$, $\neg f_i.m(s) \implies (\forall r_i \in U_i, \neg r_i.m(s))$.

The best filter is the regex lowest in the subsumption tree to subsume a set. Finding the best filters for a given set of regexes is likely to be computationally intensive, but the benefit of the filtering approach is that once good filters are found, the searching solution remains solved. All additional detail is left as an opportunity for future work.

6.2.2 Automated regex repair

Regular expression errors are common and have produced thousands of bug reports [Spishak et al. (2012)]. This provides an opportunity to introduce automated repair techniques for regular expressions. Recent approaches to automated program repair rely on mutation operators to make small changes to source code and then re-run the test suite (e.g., [Weimer et al. (2010); Le Goues et al. (2012)]). In regular expressions, it is likely that the broken regex is close, semantically, to the desired regex. Syntax changes through mutation operators could lead to big changes in behavior, but using transformations within a behaviorally similar cluster (as described in Section 4.2.1) to identify potential repair candidates could efficiently and effectively converge on a repair candidate. Semantic search can also be used in running code to find candidates, where live-running tests suites are intercepted to find the string specifications needed for a correct repair.

6.3 Fundamental Research Opportunities

6.3.1 Comparison opportunities

The analysis techniques developed in this work can be applied to a very wide variety of data sets to obtain empirical comparisons. Focusing on a particular community can obtain results that are likely to be impactful for that community.

Suggested comparisons Within one data source (like GitHub) and one programming language (like Python) or one regular expression language (like EMCAScript), different types

of projects could be compared to discover differences in usage that can be attributable to project type. Similarly holding other variables constant and allowing the 1. data source, 2. programming language or regular expression language variant, 3. project size, 4. developer maturity level, 5. file-level bug-counts, or 6. file-level time since creation to vary could reveal details about how usage of regular expressions (in context of these variables) affects the software development community. Many other comparisons are likely possible, and this is all left for future work.

Evolution of regexes   When a particular piece of code contains a regex, it is possible for that regex to be altered over the course of time, and for these alterations (visible via commit logs or similar) to indicate what details are important for a regex to cover that are often missed in early versions, as well as many possible insights into preferred representations for refactoring.

6.3.2   Extending feature analysis

Ordinary characters   This work focused mostly on features that provide string specification capabilities other than how to specify characters (i.e., the KLE and BKR feature frequencies were counted, but octal, hex and literal representation frequencies were not counted). Yet the comprehension evaluations indicated that the strongest refactoring opportunity for understandability was $\overrightarrow{T4T1}$, which is a refactoring of how to express characters. Furthermore, when comparing feature support of different languages, one substantial and unexpected difference between variants observed by the author was the difference in available escaped invisible characters. Yet these escape characters were not included in this study, with the exception of VWSP (\v). These omissions do not reduce the impact of the results presented, but provide an opportunity for additional impactful discoveries to be made in future work.

Feature set comparisons   Additionally, future research is needed into the feature sets of different variants, extending the work done in this thesis presented in Table 4.5 an Table 4.6. Several very popular variants are not represented in these tables, including the NSRegularExpression variant (used by Swift and Objecive-C), the MathWorks variant (used in MATLAB),

the D Regular expressions variant, the TRE variant, VIM Regular Expressions and several others.

Portability Guides At the time of this writing, the best resources for users wishing to port patterns from one regular expression language to another are tools like RegexBuddy (Reg, 2016). More study is needed to fully document the language details and make it possible for users to transform regular expressions across languages without accidentally changing the meaning of the regex. For example, in JavaScript and POSIX ERE, the pattern `"a\Z"` compiles to a regex matching the string `"aZ"`, because the pattern fragment `\Z` has no special significance and the backslash is ignored. In Python Regular Expressions, this fragment 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).

Ephemeral regex exploration In some environments, such as command line or text editor, regexes are used extensively by the surveyed developers, but these regular expressions do not persist (Section 4.3.2). Thus, using a repository analysis for feature usage only illustrates part of how regexes are used in practice. Exploring how the feature usage differs between environments would help inform tool developers about how to best support regex usage in context, and is left for future work.

6.3.3 Taxonomy and formal language studies

Language analysis The man page for Regex(7) says 'Having two kinds of REs is a botch.' (reg, 2009). As shown in this work, there are certainly more than two kinds of regular expression languages at this time, and the number of languages is bound to grow. Documentation on the relationships between these languages, their evolution over time, and details about their differences is sorely lacking. This presents an opportunity for language researchers to apply known techniques such as Bayesian phylogenic analysis (Kitchen et al., 2009) to regular expression languages, and to improve the general awareness of regular expression languages.

Formal containment   Because many newer regular expression languages draw their feature sets from previous variants, many feature sets of older languages may be formal subsets of newer languages. This presents an opportunity for future work in formally expressing the behavior of variants with the intent of determining where containment is possible. An engine supporting the features of all languages may not be possible, but would be a powerful tool, and so more language is needed into the formal relationship between variants.

CHAPTER 7.   CONCLUSION

7.1   Summary Of Contributions

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

The contributions of this work are:

- A survey of 18 professional software developers about their experience with regular expressions,

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

- An approach for measuring behavioral similarity of regular expressions and qualitative analysis of the most common behaviorally similar clusters, and

- An evidence-based discussion of opportunities for future work in supporting programmers who use regular expressions, including refactoring regexes, developing regex similarity analyses, and providing migration support between languages.

APPENDIX A.   FEATURE STUDY ARTIFACTS

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.

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.

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** rewinder.rewind() **do**
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.

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** fileAbsPath in pythonAbsPaths **do**
 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 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 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 13 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,

---

Listing 1 Example of sourceJSON for one citation

```
1    {
2        "data":{
3            "sha":"d2d70ff70847b171c23a8e18c7fdac5e02e15fca",
4            "commitS":"1260174268"
5        },
6        "meta":{
7            "clone_url":"https://github.com/ugtar/git-cola.git",
8            "default_branch":"master",
9            "repoID":"2098485",
10           "name":"git-cola"
11       },
12       "type":"Github"
13   }
```

---

and so after enough data had been collected, the data from all runs was merged into a single database.

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 flexible description of the source. An example is provided in Listing 1.

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 Section 4.1.1.4

regexFunction The name of the function called in the utilization.

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[1] library, which is a Python AST parser that is actively being maintained in order to support Pylint[2]. 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. Consider some source code like `re.compile("X'\)")` which compiles to `X'\)` (matching the string `"X')"`). A naive regex like `re.compile\((["']["^"']+["'])\)` would capture the string `"X"` instead of the actual pattern `"X'\)"`.

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

---

[1] https://www.astroid.org/
[2] https://www.pylint.org/

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 ghtorrent[3] 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.

Patterns Excluded From Analysis

This section contains the 114 patterns obtained from Python projects that were excluded from analysis for various reasons. Extra-long patterns are truncated, with the number of truncated characters displayed, so 20 truncated characters would be displayed like `...<20>`. These patterns are formatted like regexes to help visualize empty space or patterns, but may or may not actually be able to compile to valid regexes.

19 Patterns using non-Python or rare features

6: IFC (If conditionals)

- `^(\()?([^()]+)(?(1)\))$`

---

[3] http://ghtorrent.org/

- `(?<=[\w)\]"\']|([ ]))(===?|!==?|[<>]=?)(?=[\w({\["\']|(?(1)\b\b|[ ]))`

- `(?<=[\w)\]"\']|([ ]))(=|[-+*/%^&|]=|<<=|>>>?=)(?=[\w({\["\']|(?(1)\b\b|[ ]))`

- `(?<=[\w)\]"\']|([ ]))([-+*/%^]|&&?|\|\|?|<<|>>>?)(?=[\w({\["\']|(?(1)\b\b|[ ]))`

- `([^()]+?)\s*(\()?(\d{4})(?(2)\))$`

- `^((?:https?://)?(?:youtu\\.be/|(?:\\w+\\.)?youtube(?:-nocookie)?\\.com/)` ...<108>

3: IFEC (If-else conditionals)

- `^(?:(a)|c)((?(1)b|d))$`

- `^(?:(a)|c)((?(1)|d))$`

- `(?:(\\[)|\\.|^)((?(1)[^]]*|[^.[]*))(?(1)(?:\\]|$)([^.[]+)?)`

5: NCND (Named conditions)

- `(?P<g1>a)(?P<g2>b)?((?(g2)c|d))`

- `(?P<quote>)(?(quote))`

- `\A(?P<head>.*?)(?P<escape>\\*)(?P<symbol>\$(?P<brace_open>\{)?` ...<51>

- `\A(?P<sign>__)?(?P<is_time>T)?(?P<amount>\d+)(?P<unit>(?(is_time)[SMH]|[DW]))?\Z`

- `\\$(?P<_bracket_>\{)?((?(_bracket_)(?:\\\}|[^\}])*|(?:\$|[A-Z]+` ...<32>

2: ECOM (Comments)

- `(?:http://)?(?:\w+\.)?depositfiles.com/(?:../(?#locale))?files/(.+)`

- `\n                    (?:\\W|^)    (?# Break or beginning)\n` ...<455>

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

- `([[:alpha:]]+://)?((([[:alnum:]]+)(:[^:@]+)?@)?([^:]+)(:[[:digit:]]+)?(/.*)?`

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

- `(?P<g1>a)(?P<g2>b)?((?(g2)c|d))`

- `(?P<quote>)(?(quote))`

IFC (If conditionals) six patterns like `"(?(2)\))"`

- `^(\()?([^()]+)(?(1)\))$`

- `(?<=[\w)\]"\']|([ ]))(===?|!==?|[<>]=?)(?=[\w({\["\']|(?(1)\b\b|[ ]))`

- `(?<=[\w)\]"\']|([ ]))(=|[-+*/%^&|]=|<<=|>>>?=)(?=[\w({\[""\']|(?(1)\b\b|[ ]))`

- `(?<=[\w)\]"\']|([ ]))([-+*/%^]|&&?|\|\|?|<<|>>>?)(?=[\w({\[""\']|(?(1)\b\b|[ ]))`

- `([^()]+?)\s*(\()?(\d{4})(?(2)\))$`

- `^((?:https?://)?(?:youtu\\.be/|(?:\\w+\\.)?youtube(?:-nocookie)?\\.com/)` ...<107>

22 Patterns causing some parsing error

- `▪`
- `*`
- `\u`
- `[]`
- `[ \n]*[[]`
- `--.*[\n\Z]`
- `\here.(\w*)`

- `?(i)reftest`
- `\$\Id[^$]*\$`
- `.NET[\/\]VC7`
- `?(i)mochitest`
- `NET 2003[\/\]VC7`
- `[\ud800-\udfff]`
- `\citation\{([^\}]+)\}`

- `[^\t\n\r -~\x85\xa0-\\uD7FF\\uE000-\\uFFFD]`

- `\n     (?P<SN>.*?)              # NG(SN)` ...<520>

- `^\n\s*((?:-|\w|\&|\*)+)  # return type\n\s+` ...<166>

- `class _EOF` ...<27,347> (astroid captured a whole file)

- `\n(?P<ret>(-|\\w|\\&|\\*)+\\s*)   # return type\n\\s+` ...<170>

- `[\x00-\x08]|[\x0b-\x0c]|[\x0e-\x19]|[\ud800-\udfff]|[\ufffe-\uffff]`

- `([\ud800-\udbff](?![\udc00-\udfff])|(?<![\ud800-\udbff])[\udc00-\udfff])`

- `[\x01-\x08\x0b\x0e-\x1f\x7f-\x9f\ud800-\udfff\ufdd0-\ufdef\ufffe` ...<327>

73 Patterns requiring Unicode support to parse

- `(?u)([^\w\.\'\-\/,&])`
- `(?u)[^-\w.]`
- `(?u)\((CODE|ID)[^\)]*\)`
- `(?u)\b\w{2,25}\b`
- `(?u)\w+`

- `([-\s]+)(?u)`
- `(\$+\w*|[^\W\d]\w*)(?u)`
- `/\*.*?\*/(?us)`
- `<(.+?)(\s.*?)?>.*?</.+?>(?uism)`
- `\s+(?u)`

- `\s+(?um)`
- `\w+(?u)`
- `(?P<fn>[^\u59d3]+)\u59d3?`
- `(?u)#(x?\d+)`
- `(?u)%\(.+?\)s`
- `(?u)&(.+?);`
- `(?u)\b\w\w+\b`
- `([\u2222\u2223])`
- `([\x80-\uffff])`
- `[.\u3002\uff0e\uff61]`
- `[0-9\uff10-\uff19]{1,2}\u5206`
- `[0-9\uff10-\uff19]{1,2}\u65e5`
- `[0-9\uff10-\uff19]{1,2}\u6642`
- `[0-9\uff10-\uff19]{1,2}\u6708`
- `[0-9\uff10-\uff19]{2}\u5e74`
- `[0-9\uff10-\uff19]{4}\u5e74`
- `[\-\u30fc]`
- `[\\\"\a\b\f\r\v\x80-\uffff]`
- `[\n \u200b]`
- `[\n\u2029]`
- `[\u0430-\u044f]+`

- `[\u0430-\u044fa-z]+`
- `[\u0621\u0624\u0626]`
- `[\u064b-\u0652]`
- `[\u2014\u2018]`
- `[\u201c\u201d]`
- `[^\t\n\r -\ud7ff\ue000-\ufffd]`
- `\n[^\n\u2029]*$`
- `\u0432\u0447\u0435\u0440\u0430`
- `\u2019`
- `\u6642\u534a`
- `\u7b7e\u5230(\d+)\u5929`
- `\u8f6c\u53d1\u5fae\u535a`
- `^[\u0622\u0623\u0625]`
- `[\\\"\a\b\f\r\v\t\n\x80-\uffff]`
- `[\x00-\x1f\\\"\b\f\n\r\t\u2028\u2029]`
- `\u5348\u5f8c|PM|\uff30\uff2d|pm|PM`
- `[^\t\n\r -~\x85\xa0-\ud7ff\ue000-\ufffd]`
- `(?u)^ \t]*==[ \t]*(\w)[ \t]*==[ \t]*\n`
- `\u0440\u0443\u0441\u0441\u043a\u0438\u0439`
- `\u0441\u0435\u0433\u043e\u0434\u043d\u044f`
- `([\x20\x21\x23-\x5B\x5D-\x7E\xa0-\uffff]+)`

- `\u5e73\u6210|[Hh][0-9\uff10-\uff19]{2}\u5e74`
- `<B\s+CLASS="function"\s*>(.*?)\(\)</B\s*>(?uism)`
- `[\\’"‘\u2018\u2019\u201c\u201d\u2032\u2033\u2034]+`
- `[0-9\uff10-\uff19]{1,2}[:\uff1a][0-9\uff10-\uff19]{1,2}`
- `(?u)/\\|\/|\||\:|\?|\*|"|<|>|[^\u0000-\u001f\u007f-\u009f]/`
- `(?u)\.|\\|\/|\||\:|\?|\*|"|<|>|[^\u0000-\u001f\u007f-\u009f]`
- `[0-9\uff10-\uff19]{1,2}[/\uff0f\-\u30fc][0-9\uff10-\uff19]{1,2}`

- `\u043e\u043f\u0443\u0431\u043b\u0438\u043a\u043e\u0432`...<38>

- `[\x00-,/:-@\[-\^'\{-\xb6\xb8-\xbf\xd7\xf7\u0132-\u0133`...<2769>

- `[\x00-@\[-\^'\{-\xbf\xd7\xf7\u0132-\u0133\u013f-\u0140`...<2013>

- `\u0441\u0434\u0430\u0435\u0442\u0441\u044f \u0432 \u0441`...<86>

- `(\u0434\u043e\u043c\u0438\u043a|\u0434\u043e\u043c\u0438\u043a`...<557>

- `[0-9\uff10-\uff19]{4}[/\uff0f\-\u30fc][0-9\uff10-\uff19]{1,2}`...<40>

- `[\t +!#$%&()*\-/<=>?@\[\\\]^_{|}:;,.\u2026\u2012\u2013\u2014\u2015\xab\xbb]+`

- `\u05d4\u05e6\u05e2\u05ea ([^\(,]*)(.*?\((.*?)\))?(.*?\((.*?)\))?(.*?,(.*))?`

- `([0-9]+)/([0-9]+)/([0-9]+)[\s\S]\u05d1\u05e9\u05e2\u05d4[\s\S]([0-9]+):([0-9]+)`

- `\u05d4\u05d3\u05d9\u05d5\u05e0\u05d9\u05dd \u05d1\u05ea\u05d0\u05e8\u05d9\u05da`

- `[\s!?,\u3002\uff1b\uff0c\uff1a\u201c \u201d\uff08 \uff09\u3001\uff1f\u300a\u300b\xb7.]`

- `[\s!?,\u3002\uff1b\uff0c\uff1a\u201c \u201d\uff08 \uff09\u3001\uff1f\u300a\u300b\xb7.]`

- `[\s!?,\u3002\uff1b\uff0c\uff1a\u201c \u201d\uff08 \uff09\u3001\uff1f\u300a\u300b\xb7]`

- `[\s!?,\u3002\uff1b\uff0c\uff1a\u201c \u201d\uff08 \uff09\u3001\uff1f\u300a\u300b\xb7]+`

Description Of Studied Features

Elements

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[4]
use the special characters `.`,`^`,`$`,`*`,`+`,`?`,`{`,`}`,`[`,`]`,`(`,`)`,`|` and `\` to implement the features that allow
compact specification of sets of strings. These special characters can be escaped using the
backslash to be treated as ordinary characters, for example `zq\$` matches `"zq$"`.

---

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

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.

Elements: Character Classes

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

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

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]`.

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+)"`

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.

Operators

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.

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.

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+)))`.

Positions

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.

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.

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.

Feature Support Details

Caveats to consider when comparing feature sets

Variation among supported feature sets is not easy to define concisely. 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 4.5 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.

Availability of alternate libraries is common

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 regular expression libraries can be found which may offer slightly different syntax or optimizations for speed. Some of these libraries implement a language defined by a standard (like PCRE or ECMAScript) or offer a choice of languages. For example, the std::regex library[5] implements engines for ECMAScript Regular Expressions (the default), AWK Regular Expressions, POSIX BRE or POSIX ERE. The following libraries are alternatives to std::regex: Boost.Regex,

---

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

Boost.Xpressive, cppre, DEELX, GRETA, Qt/QRegExp and RE2. These alternative libraries are developed by hobby users and software giants alike, with RE2 [re2 (2015)] being a recent and notable alternative library developed by Google.

Choosing languages to compare

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 12 (70%) of the 17 languages listed in Table 4.1.5, (i.e. not MATLAB, Swift, Objective-C, D or SQL), the tables presented here should provide useful information.

Sources of data

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 Appendix A. 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.

Unranked feature descriptions

The following brief descriptions of unranked features set are provided to aide in understandability of Table 4.6. 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: equivalent characters varying only by accent or case

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

Determining feature support of four analysis tools

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[6]. For hampi, he set of regexes included in the test suite `lib/regex-hampi/sampleRegex` file within the hampi repository[7] 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[8]). For Rex, the feature set was collected empirically when attempting to use Rex as described in Section link.

---

[6] http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html

[7] https://code.google.com/p/hampi/downloads/list

[8] http://people.csail.mit.edu/akiezun/hampi/Grammar.html

For Automata.Z3, a file containing sample regexes[9] was examined to determine which features it supports. This may be an underestimation, as the set of patterns provided is small.

---

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

APPENDIX B.   CLUSTERING STUDY ARTIFACTS

Similarity matrix creation details

A free trial of Windows 7 was run within VMware on a macbook pro. The Rex [Veanes et al. (2010)] executable[1] requires .Net 4.5, and the similarity matrix creating program was written in C# using visual studio 2013. First the patterns for 3,582 Python regexes appearing in multiple projects were used to try and generate strings using Rex, which rejected 711 patterns. For the remaining 2,871 patterns that Rex could generate strings for, the test strings were stored in a distinct file for each regex, and delimited by a large random string (Rex often needs to generate muti-line test strings).

The filtered corpus of Rex-compatible regexes was written to a file to increase loading speed in the next step. For each regex in the filtered corpus, the test strings stored for that regex were loaded and all other regexes attempted to match those strings. Although regular expression engines usually perform a match quickly, an occasional pathological combination of regex and test string would cause the entire program to stall. The Parallel.For(...) functionality of C# was used to allow work to continue, but eventually the program had to be stopped using an interrupt. This caused incomplete rows of data which needed to be pruned by a separate program and re-calculated. All rows were verified in a final step before exporting the similarity matrix.

Markov Clustering Arguments

The mcl tool takes many arguments, with the main value, i, controlling inflation. A larger value of i will produce more, smaller clusters, and visa versa. A cutoff value p below which edges

---

[1] http://research.microsoft.com/en-us/downloads/7f1d87be-f6d9-495d-a699-f12599cea030/

are treated as zero, is also provided. A third value k can be used to customize the number of neighbor nodes to track per computation [van Dongen (2012)]. The default values for these three are 2, 0.75 and 4 respectively. Extensive experimentation comparing the contents of clusters using various values for i, p and k led to the choice of $i = 1.8$, $p = 0.75$ and $k = 83$. Under the advisement of the mcl manual, the directional edges produced by the similarity determining technique were averaged to form a symmetric edge weight matrix before clustering [van Dongen (2012)].

Top five complete clusters

This section displays the top five clusters so that a sense of the similarity of regexes within a cluster created by mcl can be obtained by the reader. Note that cluster 0 is excluded because it contains regexes that trivially match everything like `^`. The following five lists describe the number of projects that contain a regex, followed by the regex.

Cluster One

Cluster one has 39 patterns, is present in 95 projects, and the shortest regex is `..`. All regexes in this cluster will match two rather free characters, as exemplified by `..`. Taking another regex as an example, `\w+(\.\w+)*` will match any two WRD characters. The KLE modified group, `(\.\w+)*` does not strictly affect the clustering behavior because if this matches zero times, it still matches.

09	`\w+(\.\w+)*`	05	`[a-zA-Z_]\w*`
09	`([a-zA-Z0-9_$%]+)`	05	`[a-zA-Z_][a-zA-Z0-9_]*`
08	(Too long to display)	05	`[a-zA-Z_$][A-Za-z0-9_$..]*`
07	`[a-zA-Z]`	05	(Too long to display)
07	`[-a-zA-Z_@.]+`	04	`[a-z]+`
06	`^[\w\-:\$]+`	04	`([A-Z]+)`
06	`[a-zA-Z][-a-zA-Z0-9_:]*`	04	`((\d|\w|_)+)`
05	`(\w+)(.*)`	04	`([\d\w\-_]+)(.*)`

03 `(..)`   **02** `(\w+\W*)`

03 `[A-Z]+`   **02** `[a-zA-Z0-9_-]+`

03 `[a-zA-Z]+`   **02** `([_A-Za-z]\w*)`

03 `[a-zA-Z_][a-zA-Z_0-9]*`   **02** `({[^}]+})?(\w+)`

03 (Too long to display)   **02** `([$\\%&@]+)?(\w+)`

03 (Too long to display)   **02** `[a-zA-Z][a-zA-Z0-9]*`

02 `..`   **02** `[A-Za-z_][A-Za-z0-9_]*`

02 `[a-z_]`   **02** `((,|^)\s*[\w\-\.]+)`

02 `[\w ]+`   **02** `[A-Za-z0-9][A-Za-z0-9\-_]*`

02 `[\w']+`   **02** (Too long to display)

02 `^.(.*).$`   **02** (Too long to display)

02 `[A-Za-z]`

Cluster Two

Cluster two has 36 patterns, is present in 89 projects, and the shortest regex is `(\W)`. All regexes in this cluster will obtain a match with one NWRD character, or close to that. For example, `[^0-9A-Za-z]` will match the '_', which is not a NWRD character, but behaves the same for the rest of the characters. So since there are so many characters in the NWRD character class, there will be many opportunities to match and this regex will match other NWRD-like regexes most of the time.

09 `(\W+)`   **05** `[^\w]`

08 `[^A-Za-z0-9_.-]`   **05** `[^A-Za-z0-9]`

08 `[\t-\r -/:-@[-'{-~]`   **05** `[^a-zA-Z0-9_-]`

07 `[^\w]+`   **04** `[\W]+`

07 `[^a-z0-9]+`   **04** `[^\w\.\-]`

06 `[^-A-Za-z0-9_.]`   **04** `[^A-Z^a-z^0-9^:]+`

06 `[^a-zA-Z0-9._-]`   **04** `[^A-Z^a-z^0-9^\/]+`

06 `[^A-Za-z0-9_. *()-]!`   **04** (Too long to display)

03	`[^\w.-]`	**02**	`[^\w\d_]`
03	`[^a-z0-9]`	**02**	`[^0-9A-Za-z]`
03	`[^A-Za-z0-9 ]`	**02**	`[^a-zA-Z0-9.]`
03	`[^a-zA-Z.0-9]`	**02**	`[^\w\.\- ]`
02	`(\W)`	**02**	`[^\w\-\.\/]+`
02	`[\W_]`	**02**	`^\s*(\S+)\s*$`
02	`[^_\w]`	**02**	`[^A-Za-z0-9_\.\-]`
02	`[\W_]+`	**02**	`[^A-Za-z0-9_.;\-]`
02	`[^\w.-]+`	**02**	`[^A-Za-z0-9_.:;\-]`
02	`[^A-Za-z_]`	**02**	(Too long to display)

Cluster Three

Cluster 3 has 33 patterns, is present in 89 projects, and the shortest regex is `(\s)`. The regexes in this cluster will be satisfied with a match of a single space character, or close to it. Many of these regexes specify other behavior, like `[\s\-]+`, which also will match a `-` character, but since the `\s` character class has several more characters, it is favored in randomly generated strings.

09	`[ \t]+`	**05**	`[\s]+`
08	`(\s)`	**05**	`((\t)|(     ))(.*)`
08	`[-./\s]`	**04**	`[ \t]`
08	`[\-\s]`	**04**	`\s+\S`
08	`[\s\-]+`	**04**	`[ \t\n]`
08	`[\n\r\t\x0b\x0c]`	**04**	`[\s\(\)]+`
07	`,?\s+`	**03**	`[\s]`
07	`\s([^a])`	**03**	`\[|\]|\n+|\s+`
07	`([\t ]+)`	**02**	`\s|,`
07	`\s([^a]*)`	**02**	`\s.*`
06	`[\s,]+`	**02**	`\S\s`

02 `[-\s]`

02 `\s*,\s*|\s+`

02 `[,\s]+`

02 `\s+(\S+\s*)$`

02 `( +|\n)`

02 `(^package:)|\s`

02 `[ \t:]+`

02 `(\s*[;,\s]\s*)`

02 `(\s|,)+`

02 `([% \t\x80-\xff])`

02 `^\S+\s+`

Cluster Four

Cluster four has 27 patterns, is present in 74 projects, and the shortest regex is `\S+`. The regexes in this cluster will be satisfied with a single one of most non-space characters. For example, `[^0-9, ]` is exclusive of digits, and allows the tab and other whitespace. But it does match the dozens of other NWSP characters like most punctuation and letters.

09 `^[^-_.]+`

02 `\S+`

08 `^(([a-zA-F0-9_-]*)[\.]?.*$`

02 `^.+$`

05 `[^a-z]`

02 `^\S`

04 `(\S+)\s*(.*)`

02 `(\S+)`

04 `^[ \t]*[^# \t]`

02 `[^A-Z]`

03 `[^\d]`

02 `[^0-9]+`

03 `[^0-9, ]`

02 `\d*(\S)`

03 `^([^ ]+)`

02 `^(\S+):?`

03 `^([^:]+)$`

02 `[^0-9:\.]`

03 `^ *(.*) *( *) *$`

02 `<[^>]+>|[^\s<]+`

03 `"([^"]+)"|(\S+)`

02 `^(\S+ \S+ \S+ \S+: )?(.*)$`

03 `^([^<]*)(/(<.*))?$`

02 `([\x20\x21\x23-\x5B\x5D-\x7E]+)`

03 `^(\s*)()(\S+)(.*$)`

02 `\s*([^\s\(\)"\\'=,\[\]/\?]+)\s*`

03 `((([^-/]+)-?([^/]+)?)/?.*`

Cluster Five

Cluster five has 23 patterns, is present in 58 projects, and the shortest regex is `\d`. The regexes in this cluster will tend to match digits, even if they might match other characters as well like `([0-9]+)([DdHhMm]|[sS]?)`, which actually requires a non-digit to match as well. This regex may have been clustered with the rest of these regexes because test strings generated to match it must include a digit, and so the pure-digits will match it 100% of the time.

08 `\d`

08 `([0-9.]+)(\S*)$`

07 `[\.0-9]+`

04 `[0-9]`

04 `([0-9.]+)`

04 `(\d+(\.\d+)*)`

03 `(\d)`

03 `([\d\.]+)`

03 `[0-9a-fA-F]{2}`

03 `\s*(\d+)\s*`

03 `([0-9]+)([DdHhMm]|[sS]?)`

03 (Too long to display)

02 `[\d\.]+`

02 `[+-]?\d+`

02 `[-+]?\d+`

02 `(\d+)(.*)`

02 `[1-9][0-9]*`

02 `[{ ]*[-+0-9]`

02 `\d+(\.\d*)?`

02 `([+-]?[\d\.]+)(\S+)`

02 `([idel])|(\d+):|(-?\d+)`

02 `([\-0-9])|([\-0-9]/[0-9])`

02 `(0x[0-9A-Fa-f]+|0\d*|[1-9]\d*)`

APPENDIX C.   SURVEY ARTIFACTS

Survey Questions

The survey given to Dwolla developers (Chapter 4.3) is presented on eight pages in Figure C.1, Figure C.2, Figure C.3, Figure C.4, Figure C.5, Figure C.6, Figure C.7 and Figure C.8.

The survey given to Dwolla developers (Chapter 4.3) is presented on eight pages in Figure C.1, Figure C.2, Figure C.3, Figure C.4, Figure C.5, Figure C.6, Figure C.7 and Figure C.8.

	General purpose	scripting	query languages	command line	text editor	other
101+	1	1	0	3	3	0
51-100	1	0	0	3	2	0
21-50	1	2	1	0	1	1
11-20	3	3	0	2	5	0
6-10	5	3	0	3	0	0
1-5	6	4	2	5	5	0
0	1	5	15	2	5	0

Table C.1   Q4: Please estimate the N regex you compose per year (by technical environment).

	capturing	counting lines	counting all	finding	filtering	single char	parse user input	parse generated	other
v. freq	1	1	1	3	0	0	2	2	0
freq.	9	2	3	7	1	0	5	1	1
occ.	3	5	4	3	8	1	5	4	0
rarely	5	3	3	4	2	3	3	3	0
v. rarely	0	3	4	1	5	5	3	5	1
never	0	4	3	0	2	9	0	3	16
	3.3	2.0	2.2	3.4	2.1	0.8	3	2.1	0.3

Table C.2   Q5: Please describe how often you compose regex for a particular problem type.

Regex usage in practice

This survey is to collect information for a research project in analyzing regex usage in practice. As this is for research, please answer as accurately as possible, not guessing what is the desired answer and providing that.

This survey will be available until Thursday, August 14 at 5:00pm CST.

Participants who complete the survey will be entered in a drawing to receive a gift certificate to Target.

Your username will be recorded when you submit this form.
* Required

1. **I am a professional software developer/maintainer.** *
   (one year or more experience developing and/or maintaining software)
   Mark only one oval.

   ◯ True

   ◯ False   *After the last question in this section, skip to "Thanks for your participation!."*

2. **How many years of programming/maintenance experience do you have?** *
   (rounding up if not sure)

   ....

3. **I have used regular expressions (regex) in a work environment.** *
   Mark only one oval.

   ◯ True

   ◯ False   *Skip to "Thanks for your participation!."*

Quantifying my usage of regex in a work environment

Figure C.1   Page 1 (of 8) from the survey deployed to professional developers

4. **Please estimate the N regex you compose per year (by technical environment).** *
   Mark only one oval per row.

	0	1-5	6-10	11-20	21-50	51-100	101+
General purpose languages (Java, C++, C#, etc.)	◯	◯	◯	◯	◯	◯	◯
scripting languages (Bash, Perl, Python)	◯	◯	◯	◯	◯	◯	◯
query languages (sql and similar)	◯	◯	◯	◯	◯	◯	◯
command line search (grep, awk, sed, find, etc.)	◯	◯	◯	◯	◯	◯	◯
searching within a text editor (IntelliJ, TextWrangler, Vim, etc.)	◯	◯	◯	◯	◯	◯	◯
other	◯	◯	◯	◯	◯	◯	◯

5. **Please describe how often you compose regex for a particular problem type.** *
   (regardless of technical environment)
   Mark only one oval per row.

	never	very rarely	rarely	occasionally	frequently	very frequently
capturing parts of strings	◯	◯	◯	◯	◯	◯
counting the number of lines that match a pattern	◯	◯	◯	◯	◯	◯
counting the number of substrings total that match a pattern	◯	◯	◯	◯	◯	◯
locating content within a huge file or files	◯	◯	◯	◯	◯	◯
filtering collections (lists, tables, etc)	◯	◯	◯	◯	◯	◯
checking for the existence of a single character	◯	◯	◯	◯	◯	◯
parsing user input	◯	◯	◯	◯	◯	◯
parsing generated text	◯	◯	◯	◯	◯	◯
other	◯	◯	◯	◯	◯	◯

6. **If you selected 'other' in either or both of the above 2 questions, please explain below what you are indicating**

....

....

....

....

....

7. **Overall, I compose about N regex per year.**
   *
   (please fill in the text box with an integer)

....

Figure C.2   Page 2 (of 8) from the survey deployed to professional developers

8. **On average, I go about N days without using regex.** *

....

Usage frequency of select features

9. **How often do you use endpoint anchors?** *
   example: with input "I like my sandwich", the regex 'my .*' can find a match, but the regex '^my .*$' does not find a match, because it uses endpoint anchors: '^' and '$'
   Mark only one oval.

   ( ) never

   ( ) very rarely

   ( ) rarely

   ( ) occasionally

   ( ) frequently

   ( ) very frequently

10. **How often do you use capture groups?** *
    example: with input "my name is mud", the regex 'my name is (.*)' will capture 'mud'
    Mark only one oval.

    ( ) never

    ( ) very rarely

    ( ) rarely

    ( ) occasionally

    ( ) frequently

    ( ) very frequently

11. **How often do you use look-aheads, negative look-aheads, look-behinds or negative look-behinds?** *
    example: with input "xyz", the regex '(?<!x)yz' will fail to find a match because x comes before yz (it's a negative look-behind because of the '!' character. A regex with positive look-ahead looks like: 'a(?=bc)')
    Mark only one oval.

    ( ) never

    ( ) very rarely

    ( ) rarely

    ( ) occasionally

    ( ) frequently

    ( ) very frequently

Figure C.3   Page 3 (of 8) from the survey deployed to professional developers

12. **How often do you use lazy repetition?** *
example: with input "<g>good</g>bad" the regex '(<.+?>)' will onlycapture "
<g>good</g>" , whereas the regex '(<.+>)' will capture the whole input. (notice the '?'
character in the first regex)
Mark only one oval.

◯ never

◯ very rarely

◯ rarely

◯ occasionally

◯ frequently

◯ very frequently

13. **How often do you use word boundaries?** *
example: with input "smooshedwords", the regex '\bwords' will not match because the cursor
between 'smooshed' and 'words' sees a word character on both sides. With input "spaced
words", the same regex will match because the space is a non-word character.
Mark only one oval.

◯ never

◯ very rarely

◯ rarely

◯ occasionally

◯ frequently

◯ very frequently

Your preferences

14. **Do you prefer to use custom character classes or default character classes more
often?** *
example of a custom character class: '[0-9]' and of a default character class: '\d' (both will
match digits 0-9)
Mark only one oval.

◯ use only default

◯ use default more than custom

◯ use both equally

◯ use custom more than default

◯ use only custom

Figure C.4   Page 4 (of 8) from the survey deployed to professional developers

15. **Why do you prefer that?** *

....
....
....
....
....

16. **To solve a small parsing problem would you prefer to write a regex or write a parser in a general purpose language?** *

    example: an extremely simple parser in java might look like 's.startsWith('myPattern')' and a similar regex could be '^myPattern'
    Mark only one oval.

    ⃝ use only regex

    ⃝ use only custom parser

    ⃝ it depends

17. **If you chose 'it depends', please explain why.**

....
....
....
....
....

18. **If you know it won't affect your use case would you prefer to use '.*' or '.+' ?** *

    example: you are capturing the content between two commas. You know there will always be content between the commas in your input. Would you rather write ',.*,' or ',.+,' ?
    Mark only one oval.

    ⃝ always use '.*'

    ⃝ use '.*' more than '.+'

    ⃝ use both equally

    ⃝ use '.+' more than '.*'

    ⃝ always use '.+'

19. **Why do you prefer that?** *

....
....
....
....
....

Figure C.5   Page 5 (of 8) from the survey deployed to professional developers

20. **Assuming you need to use back-references, would you prefer to use numbered or named back-references?** *

    example: with input "aaabaabab" and regex '(a+)b\1', the 'aabaa' portion will be matched (also the later 'aba' portion). You could do the same thing with the regex '(?P<ayes>a+)b(?P=ayes)' which uses named back-references.
    Mark only one oval.

    ◯ always use numbered backreferences

    ◯ always use named backreferences

    ◯ it depends

21. **If you chose 'it depends', please explain why.**

    ....

    ....

    ....

    ....

    ....

Testing code with regex

22. **In General, how often do you write tests for your code?** *
    Mark only one oval.

    ◯ never

    ◯ very rarely

    ◯ rarely

    ◯ occasionally

    ◯ frequently

    ◯ very frequently

    ◯ always

23. **How often do you write tests for your regexes?** *
    Mark only one oval.

    ◯ never

    ◯ very rarely

    ◯ rarely

    ◯ occasionally

    ◯ frequently

    ◯ very frequently

    ◯ always

Figure C.6   Page 6 (of 8) from the survey deployed to professional developers

24. **What tools do you use, if any, to help compose/test your regexes?** *
    (please write 'None' if you never use any tools to help)

    ....

    ....

    ....

    ....

    ....

User experiences

25. **I have used the options wrapper** *
    example: the regex '(?i)caps' will be case insensitive because of the '(?i)'
    Mark only one oval.

    ( ) True

    ( ) False

26. **I have used regex to parse HTML or XML** *
    Mark only one oval.

    ( ) True

    ( ) False

27. **What pain points have you encountered with regular expressions?** *
    (this can be anything)

    ....

    ....

    ....

    ....

    ....

28. **What do you use the 'word character' default character class '\w' for?** *
    (it is equal to the custom character class '[a-zA-z_]')

    ....

    ....

    ....

    ....

    ....

Figure C.7   Page 7 (of 8) from the survey deployed to professional developers

29. **I have used the repetition range modifier** *
example: a{4,12} will match the letter 'a' repeated between 4 and 12 times
Mark only one oval.

⬭ True

⬭ False

30. **Do you have anything else to add about your experiences with regexes?** *

....

....

....

....

....

Thanks for your participation!

☐ Send me a copy of my responses.

Powered by

Google Forms

Figure C.8   Page 8 (of 8) from the survey deployed to professional developers

	STR or END	CG	any of: LKA NLKA LBK NLKB	LZY	WNW
very frequently	1	2	0	0	1
frequently	9	4	1	2	3
occasionally	5	9	2	6	6
rarely	2	2	5	2	2
very rarely	1	1	7	4	6
never	0	0	3	4	0
avg	6.1	5.8	3.5	4	4.8
	18	18	18	18	18

Table C.3   Q9 - Q13: Usage frequency of select features

	What pain points have you encountered with regular expressions?
A	long ones can be hard to read
B	
C	Maintainability. Readability by other developers.
D	inconsistencies between implementations. Some regexes work differently (or don't work) in some languages.
E	
F	Hard to read/debug. Easy to have edge cases.
G	It's farther away from English than most of the programming languages I use.
H	Readability. Edge cases.
I	Differences in implementation across languages
J	Capturing too much information, behaving unexpectedly due to not thinking about all scenarios.
K	Hard to always remember best practices.
L	Before I found the online tools, it was very difficult to write them since I've never read up on them. With the tools, I can usually hack something together.
M	Composition of multiple groups
N	Not every system supports all the functionality of regex and knowing what is supported across various languages is tricky. It is terrible to read (especially later after initial development) and expose what it was meant to do without very verbose method names. Amazingly bad for long term maintenance, and they are often difficult to test for edge cases without being very careful or robust in your testing.
O	I don't use them often enough to have the syntax perfectly memorized. Sometimes there is trickiness to getting the expression right. Typically once they are written, especially by someone else, they can be difficult to interpret afterwords.
P	certain cases they can get long and unwieldy for my meager brain.
Q	Creating the correct one to match the input
R	Learned the hard way that it should never be used for html. Also, readability is another big pain point. Almost always need to extract an intent revealing method.
S	Can be hard to read after the fact
T	I have trouble with forward and back references. Regex has its place and may not be suitable for everything.

Table C.4   Q27: What pain points have you encountered with regular expressions?

APPENDIX D.   Equivalence class artifacts

Complete Node Membership Descriptions

D.0.0.1   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 4.7, 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.

D.0.0.2 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.

D.0.0.3   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.

D.0.0.4   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.

D.0.0.5   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).

APPENDIX E.   COMPREHENSION STUDY ARTIFACTS

Regexes And Matching Strings Tested On Mechanical Turk, Organized By Metagroup

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]`
	`"g%a9"`		`"&deez"`		`"fall"`
	`"%-F"`		`"t&dazz"`		`"afmon"`
	`"0123abC"`		`"&diez"`		`"fanopster"`
	`"%0G"`		`"&dazez"`		`"infalobl"`
	`"%8F-1"`		`"douz"`		`"famlnk"`

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]`
	`"11.3"`		`"1x1g1333%"`		`"d912a"`
	`"12."`		`"Lxg134%"`		`"h12f"`
	`"888"`		`"1492%"`		`"aff321"`
	`"0a.2"`		`"xg13%"`		`"123af"`
	`".075"`		`"xg1345%2"`		`"aaa4a"`

Metagroup 3: testing C1 vs C4, focusing on WRD

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

C4 `[&(\w+);]` C4 `[1q\w\w]` C4 `[tuv\w]`

`"&&"`	`"1q&&"`	`"tuv\w"`
`"abc_;"`	`"1aqabc_"`	`"tuv&"`
`"&&a_9;"`	`"1qabc2"`	`"tuvx"`
`"&aFF;"`	`"a1q245"`	`"amtuv0"`
`"&a-F;"`	`"1q\w\w"`	`"pqtuv"`

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]`

`"abc"`	`"84732211"`	`"ggg"`
`"."`	`"axb33"`	`"    l"`
`"*1"`	`"*1"`	`"el ela"`
`"123"`	`"123"`	`"tp11"`
`"}x"`	`"}x"`	`"0123abC"`

Metagroup 5: testing L2 vs L3 (note that the pair `\..*` and `\.+` 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+`

`"99"`	`"qtmnzba"`	`"98"`
`"..."`	`"qtzaaa"`	`"ROR"`
`"a dog."`	`"za"`	`"ARROW"`
`"."`	`"azazaza"`	`"qRs"`
`"abc"`	`"az"`	`"qrs"`

Metagroup 6: testing T1 vs T3

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

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

   `"${881:}"`   `"t..5*"`   `"{$88.\}"`

   `"{12:-}"`   `"ampty.*$0"`   `"{$0.3}"`

   `"${09.1::}"`   `"sit."`   `"$99.2"`

   `"${31:13}"`   `"t.$111*"`   `"{$31.13}"`

   `"#${1:x22}"`   `"qt.$$$41*"`   `"{$112.4}"`

Metagroup 7: testing D1 vs D2 vs D3

D1 `((q4f){0,1}ab)`   D1 `(dee(do){1,2})`

D2 `((q4f)?ab)`   D2 `(deedo(do)?)`

D3 `(q4fab|ab)`   D3 `(deedo|deedodo)`

   `"ab"`   `"do deedodeedo"`

   `"fq4f"`   `"dodeedee do"`

   `"xyzq4fab"`   `"do deedodo"`

   `"zlmab"`   `"dedoode"`

   `"qfa4"`   `"deedo do"`

Metagroup 8: testing C1 vs C2 vs C5

C1 `tri[a-f]3`   C1 `no[w-z]5`

C2 `tri[abcdef]3`   C2 `no[wxyz]5`

C5 `tri(a|b|c|d|e|f)3`   C5 `no(w|x|y|z)5`

   `"tri3def"`   `"nov5"`

   `"triabc3"`   `"noxy5"`

   `"tric3"`   `"now5"`

   `"trig3"`   `"ny5"`

   `"abc3"`   `"noz"`

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

C2/T1 `([}{])`   C2/T1 `([:;])`

C5/T1 `(\{|\})`   C5/T1 `(:|;)`

C2/T4 `([\072\073])`   C2/T4 `([\0175\0173])`

    `"{o0ps"`   `";o0ps"`

    `"”__`   `"”__`

    `"{x}"`   `":x"`

    `"([c])"`   `"([c])"`

    `"pcm}"`   `"pcm;:"`

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

C1/T2 `xyz[\x5b-\x5f]`   C1/T2 `t[\x3a-\x3b]+p`

C1/T4 `xyz[\0133-\0140]`   C1/T4 `t[\072-\073]+p`

C2/T1 `xyz[_\[\]‘\^\\]`   C2/T1 `t[:;]+p`

    `"xyz_1"`   `"t;;p"`

    `"yzx’3"`   `"t}p"`

    `"xyzyx"`   `"t\73p"`

    `"xyz\133"`   `"t:;:p"`

    `"xyz139"`   `"t::;:"`

Table E.1   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

Unsure results

Participants were able to select *unsure* when they were not sure if a string would be matched by a regex (Figure 4.8). From a comprehension perspective, this indicates some level of confusion and is worth exploring.

Unsure results   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 4.21. These nodes and their average number of unsure responses are organized in Table E.1. 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.

Mapping edge indices to regex pairings

E1: T1 – T4

T1 `([}{])`   T1 `([:;])`

T4 `([\0175\0173])`   T4 `([\072\073])`

E2: D2 vs D3

D2 `((q4f)?ab)`   D2 `(deedo(do)?)`

D3 `(q4fab|ab)`   D3 `(deedo|deedodo)`

E3: C2 – C5

C2 `tri[abcdef]3`   C5 `tri(a|b|c|d|e|f)3`

C2 `no[wxyz]5`   C5 `no(w|x|y|z)5`

C2 `([}{])`   C5 `(\{|\})`

C2 `([:;])`   C5 `(:|;)`

E4: C2 – C4

C2 `[\t\r\f\n ]`

C4 `[\s]`

E5: L2 – L3

L2 `zaa*`   L2 `RR*`

L3 `za+`   L3 `R+`

E6: D1 vs D2

D1 `((q4f){0,1}ab)`   D1 `(dee(do){1,2})`

D2 `((q4f)?ab)`   D2 `(deedo(do)?)`

E7: C1 – C2

C1 `tri[a-f]3`   C1 `no[w-z]5`

C2 `tri[abcdef]3`   C2 `no[wxyz]5`

E8: T2 – T4

T2 `xyz[\x5b-\x5f]`   T2 `t[\x3a-\x3b]+p`

T4 `xyz[\0133-\0140]`   T4 `t[\072-\073]+p`

E9: C1 – C5

C1 `tri[a-f]3`   C1 `no[w-z]5`

C5 `tri(a|b|c|d|e|f)3`   C5 `no(w|x|y|z)5`

E10: T1 – T3

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

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

E11: D1 vs D3

D1 `((q4f){0,1}ab)`   D1 `(dee(do){1,2})`

D3 `(q4fab|ab)`   D3 `(deedo|deedodo)`

E12: C1 – C4

C1 `([0-9]+)\.([0-9]+)`   C4 `(\d+)\.(\d+)`

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

C1 `[a-f]([0-9]+)[a-f]`   C4 `[a-f](\d+)[a-f]`

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

C1 `1q[A-Za-z0-9_][A-Za-z0-9_]` C4 `1q\w\w`

C1 `tuv[A-Za-z0-9_]`   C4 `tuv\w`

E13: C3 – C4

C3 `[^0-9A-Za-z]`   C3 `[^0-9]`

C4 `[\W_]`   C4 `[\D]`

E14: S1 – 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]`

Qualifying Test

Regex Qualification

what does the regex 'a+' mean?

- ○ the literal sequence of 2 characters: 'a+'
- ○ any character after 'a' in the alphabet
- ○ one or more 'a' characters

what does the regex '(r|z)' mean?

- ○ any character in the range between r and z
- ○ either an 'r' or 'z' character
- ○ the literal sequence of 5 characters: '(r|z)'

what does the regex '\d' mean?

- ○ any digit character like '1' or '8'
- ○ the literal sequence of 2 characters: '\d'
- ○ the invisible 'down' character

what does the regex 'q*' mean?

- ○ zero or more 'q' characters
- ○ any line that starts with a 'q' character
- ○ the literal sequence of 2 characters 'q*'

what does the regex '[p-s]' mean?

- ○ the literal sequence of 5 characters: '[p-s]'
- ○ captures 'p' then any character, then 's'
- ○ any character in the range between p and s

Figure E.1   The qualification test taken to participate in the regex understandability study. Four out of five questions must be answered correctly.

Template

Instructions

This project has 10 subtasks, each with 6 questions labeled A–F.

For each subtask, you are presented with one regular expression (regex) pattern and 5 strings.
For each of the 5 strings, indicate whether any substring (including the entire string) matches the given pattern. The entire string does not have to match, for example if you are given the simple regex 'ab+c', the string 'xyzabbc' will match (you should select 'yes') because the substring 'abbc' matches, even though the given string starts with 'xyz'. But the string 'xyzac' will not match (you should select 'no'), because no substring can be found that matches the pattern. This is true because the regex 'ab+c' requires at least one 'b' character between a and c.

If you are unsure, you can select 'Unsure', but make a good-faith effort.

After entering your answer for each string, please compose a string of your own which contains a substring that matches the given regex pattern (part F). This string must be different from the five provided strings for the regex. Accuracy on this part, or near accuracy, is required for payment.

Regexes are shown 'raw', that is with backslashes unescaped. For example, in practice, you may need to escape the backslash used in a regex by writing '\\d', but we will show this as '\d'.

Both regexes and strings are surrounded by single-quotes for clarity. These outermost single quotes are never part of the regex or string

Please do not use any tools or write programs to inform your answers – only use what you know about regexes. Please try to spend no more than 1 minute on any subtask.

At the end of the project there is a brief survey, which also must be completed for payment.

Subtask 1. Regex Pattern: ${ST1_regex}

1.A	${ST1A}	⚪ matches	⚪ not a match	⚪ unsure
1.B	${ST1B}	⚪ matches	⚪ not a match	⚪ unsure
1.C	${ST1C}	⚪ matches	⚪ not a match	⚪ unsure
1.D	${ST1D}	⚪ matches	⚪ not a match	⚪ unsure
1.E	${ST1E}	⚪ matches	⚪ not a match	⚪ unsure

1.F Compose your own string that contains a match: [_____]

Subtask 2. Regex Pattern: ${ST2_regex}

2.A	${ST2A}	⚪ matches	⚪ not a match	⚪ unsure
2.B	${ST2B}	⚪ matches	⚪ not a match	⚪ unsure
2.C	${ST2C}	⚪ matches	⚪ not a match	⚪ unsure
2.D	${ST2D}	⚪ matches	⚪ not a match	⚪ unsure
2.E	${ST2E}	⚪ matches	⚪ not a match	⚪ unsure

2.F Compose your own string that contains a match: [_____]

Figure E.2 Template for one HIT(page 1 of 4). Red values like ${ST1_regex} are populated with regexes, and black values like ${ST1A} are populated with matching strings.

Subtask 3. Regex Pattern: ${ST3_regex}

3.A	${ST3A}	○ matches ○ not a match	○ unsure
3.B	${ST3B}	○ matches ○ not a match	○ unsure
3.C	${ST3C}	○ matches ○ not a match	○ unsure
3.D	${ST3D}	○ matches ○ not a match	○ unsure
3.E	${ST3E}	○ matches ○ not a match	○ unsure

3.F Compose your own string that contains a match: _____

Subtask 4. Regex Pattern: ${ST4_regex}

4.A	${ST4A}	○ matches ○ not a match	○ unsure
4.B	${ST4B}	○ matches ○ not a match	○ unsure
4.C	${ST4C}	○ matches ○ not a match	○ unsure
4.D	${ST4D}	○ matches ○ not a match	○ unsure
4.E	${ST4E}	○ matches ○ not a match	○ unsure

4.F Compose your own string that contains a match: _____

Subtask 5. Regex Pattern: ${ST5_regex}

5.A	${ST5A}	○ matches ○ not a match	○ unsure
5.B	${ST5B}	○ matches ○ not a match	○ unsure
5.C	${ST5C}	○ matches ○ not a match	○ unsure
5.D	${ST5D}	○ matches ○ not a match	○ unsure
5.E	${ST5E}	○ matches ○ not a match	○ unsure

5.F Compose your own string that contains a match: _____

Subtask 6. Regex Pattern: ${ST6_regex}

6.A	${ST6A}	○ matches ○ not a match	○ unsure
6.B	${ST6B}	○ matches ○ not a match	○ unsure

Figure E.3   Template for one HIT (page 2 of 4). Red values like ${ST1_regex} are populated with regexes, and black values like ${ST1A} are populated with matching strings.

6.C	${ST6C}	◯ matches	◯ not a match	◯ unsure
6.D	${ST6D}	◯ matches	◯ not a match	◯ unsure
6.E	${ST6E}	◯ matches	◯ not a match	◯ unsure

6.F Compose your own string that contains a match: [_____]

Subtask 7. Regex Pattern: ${ST7_regex}

7.A	${ST7A}	◯ matches	◯ not a match	◯ unsure
7.B	${ST7B}	◯ matches	◯ not a match	◯ unsure
7.C	${ST7C}	◯ matches	◯ not a match	◯ unsure
7.D	${ST7D}	◯ matches	◯ not a match	◯ unsure
7.E	${ST7E}	◯ matches	◯ not a match	◯ unsure

7.F Compose your own string that contains a match: [_____]

Subtask 8. Regex Pattern: ${ST8_regex}

8.A	${ST8A}	◯ matches	◯ not a match	◯ unsure
8.B	${ST8B}	◯ matches	◯ not a match	◯ unsure
8.C	${ST8C}	◯ matches	◯ not a match	◯ unsure
8.D	${ST8D}	◯ matches	◯ not a match	◯ unsure
8.E	${ST8E}	◯ matches	◯ not a match	◯ unsure

8.F Compose your own string that contains a match: [_____]

Subtask 9. Regex Pattern: ${ST9_regex}

9.A	${ST9A}	◯ matches	◯ not a match	◯ unsure
9.B	${ST9B}	◯ matches	◯ not a match	◯ unsure
9.C	${ST9C}	◯ matches	◯ not a match	◯ unsure
9.D	${ST9D}	◯ matches	◯ not a match	◯ unsure
9.E	${ST9E}	◯ matches	◯ not a match	◯ unsure

9.F Compose your own string that contains a match: [_____]

Figure E.4   Template for one HIT (page 3 of 4).. Red values like ${ST1_regex} are populated with regexes, and black values like ${ST1A} are populated with matching strings.

Subtask 10. Regex Pattern: ${ST10_regex}

10.A	${ST10A}	○ matches	○ not a match	○ unsure
10.B	${ST10B}	○ matches	○ not a match	○ unsure
10.C	${ST10C}	○ matches	○ not a match	○ unsure
10.D	${ST10D}	○ matches	○ not a match	○ unsure
10.E	${ST10E}	○ matches	○ not a match	○ unsure

10.F Compose your own string that contains a match:

Demographic Info

What is your gender?

○ Male
○ Female
○ Prefer not to say

What is your age?

Which of the following best describes your highest achieved education level?

- select one -

Do you have programming experience such as work experience or a degree in IT, computer science, or a related field?

○ Yes
○ No

Prior to completing this HIT, how familiar are you with regular expressions?

- select one -

How many regular expressions do you compose per year?

How many regular expressions (not written by you) do you read per year?

In what context do you usually read regular expressions?

Figure E.5   Template for one HIT (page 4 of 4).. Red values like ${ST1_regex} are populated with regexes, and black values like ${ST1A} are populated with matching strings.

BIBLIOGRAPHY

(1994). *IEEE Std 1003.2-1992/INT*. IEEE.

(2001). Perl timeline.

(2003). *Oracle Database SQL Reference*. Oracle.

(2004). *sed, a stream editor*. Free Software Foundation, Inc.

(2009). *REGEX(7) Linux Programmer's Manual*. man7.org.

(2015).

(2015). *git(1) Manual Page*. kernel.org.

(2015). *GNU Grep 2.24 Manual*. Free Software Foundation, Inc.

(2015). Perl history.

(2016). *CRONTAB(5) Manual Page*. man7.org.

(2016). *FIND(1) Manual Page*. man7.org.

(2016). *The MongoDB 3.2 Manual*. MongoDB, Inc.

(2016). *MySQL 5.7 Reference Manual*. MySQL.

(2016). Perl 5.10 release notes.

(2016). *PostgreSQL 9.5.2 Documentation*. The PostgreSQL Global Development Group.

(2016). Regexbuddy.

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

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340.

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

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

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

at Bell Labs, T. K. L. (1971). Text matching algorithm, u. s. patent 3568156. patent.

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

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

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

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

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

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

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

Chamberlin, D. D. and Boyce, R. F. (1974). Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA. ACM.

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

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

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

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

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

Friedl, J. (2006). *Mastering Regular Expressions.* O'Reilly Media, Inc.

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

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

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

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

Hermans, Felienne; Aivaloglou, E. (2016). Do code smells hamper novice programming? under review, TUD-SERG-2016-006.

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

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

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.

Hume, A. (1988). A tale of two greps. *Softw. Pract. Exper.*, 18(11):1063–1072.

Johnson, S. C. (2006). *Yacc: Yet Another Compiler-Compiler.* AT&T Bell Laboratories.

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

Kirrage, J., Rathnayake, A., and Thielecke, H. (2013). *Network and System Security: 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, chapter Static Analysis for Regular Expression Denial-of-Service Attacks, pages 135–148. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kitchen, A., Ehret, C., Assefa, S., and Mulligan, C. J. (2009). Bayesian phylogenetic analysis of semitic languages identifies an early bronze age origin of semitic in the near east. *Proceedings of the Royal Society of London B: Biological Sciences*, 276(1668):2703–2710.

Kleene, S. C. (1951). Representation of events in nerve nets and finite automata. Technical Report RM-704, RAND Corporation, Santa Monica, CA.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. *Automata Studies*.

Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., and Turner, J. (2006). Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.*, 36(4):339–350.

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

Lee, J., Pham, M.-D., Lee, J., Han, W.-S., Cho, H., Yu, H., and Lee, J.-H. (2010). Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth*

International Workshop on Data and Text Mining in Biomedical Informatics, DTMBIO '10, pages 23–30, New York, NY, USA. ACM.

Lesk, M.E., S. E. (2006). *Lex - A Lexical Analyzer Generator*.

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

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

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

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133.

McIlroy, M. D. (1987). *A Research UNIX Reader*. AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

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

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

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

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

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2015). Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 585–596.

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

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

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

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

Ritchie, D. M. and Thompson, K. (1974). The unix time-sharing system. *Commun. ACM*, 17(7):365–375.

Sommer, R. and Paxson, V. (2003). Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 262–271, New York, NY, USA. ACM.

Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA. ACM.

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

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

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

Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422.

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

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

van Dongen, S. (2012). *MCL manual.* micans.org, 12-068 edition.

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

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

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

Yu, F., Chen, Z., Diao, Y., Lakshman, T. V., and Katz, R. H. (2006). Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, pages 93–102, New York, NY, USA. ACM.