Readability and Refactoring of Regular Expressions

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

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

ABSTRACT

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

1. INTRODUCTION

Our contributions are as follows:

- Defined regex refactoring categories for understandability backed by empirical evidence
- Identified 3 or so other regex refactorings categories and specific instances that are worthy of further investigations
- Identified a few regex refactorings that can be eliminated because both options are equally readable

The rest of this paper is organized as follows:
Related work, study, results, discussion, conclusion.

2. RELATED WORK

TODO.MID: We are building on the survey that indicates regexes are hard to read, and the apparent lack of any regex readability refactoring attempts. Many papers have talked about refactoring, basically it is changing the form but not the behavior.

Prior work that surveyed developers about regex usage found that in a small software company, the 18 surveyed developers compose an average of 172 regexes per year. This is 48% higher than the number of regexes composed annually by MTurk participants in this work, which may be due to the nature of the jobs performed by the two populations.

3. REFACTORINGS

After studying thousands of regexes from thousands of Python projects, we have defined a set of equivalence classes for regexes with refactorings that can transform among members in the classes. Figure 1 represents X equivalence classes in grey boxes and various representations of a regex that are all semantically equivalent. The lines between the representations define possible refactorings. The directions of arrows in the possible refactorings will be discussed in Section 5 based on frequency of occurrence in the community and the understandability of the regexes using 180 study participants.

Next, we describe each group in detail:

CCC Group.

DBB Group.

LIT Group.

LWB Group.

SNG Group.
This equivalence class contains three representations of a regex that deal with repetition of a single element in the regex, represents by `S`. The representation in *S1*, `S{3}`, states that S appears exactly three times in sequence. This can alternately be expressed with a double-bound where the upper and lower bounds are the same, as in *S3*. Removing the repetition symbols in either *S1* or *S3* yields *S2* which is represented explicitly as `SSS`.

Using an example from a Python project, the regex '`[^ ]*\.[A-Z]{3}`' is a member of *S1* and could be refactored to *S3* as '`[^ ]*\.[A-Z]{3,3}`' or to *S2* as '`[^ ]*\.[A-Z][A-Z][A-Z]`', depending on programmer preferences.

4. STUDY

After defining the possible regex refactorings as described in Section 3, we wanted to know which representations in the equivalence classes are considered smelly and which are considered desirable. Desirable for regexes can be defined many ways, including maintainable and understandable. As prior work has shown that regexes are difficult to read [], we seek to define refactorings toward understandability.

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

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

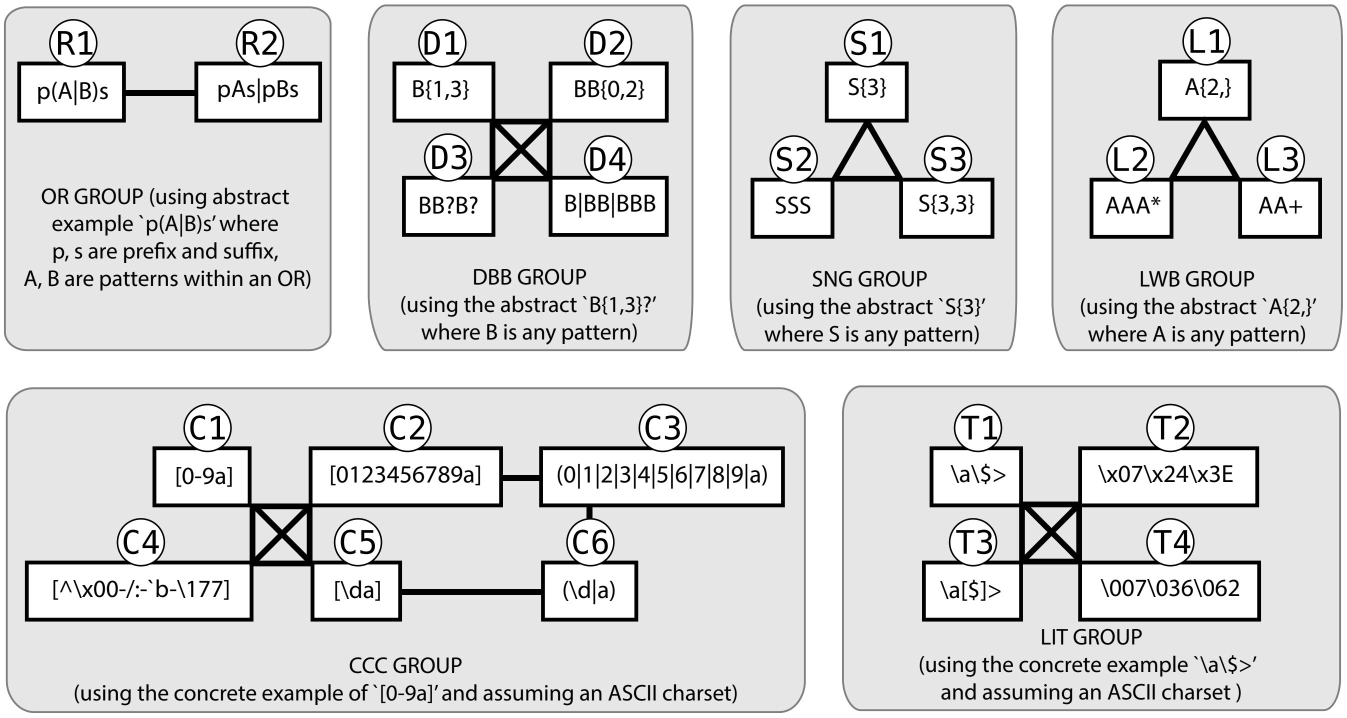
R1
R2
p(A|B)s
pAs|pBs
OR GROUP (using abstract example `p(A|B)s' where p, s are prefix and suffix, A, B are patterns within an OR)
D1
D2
D3
D4
B{1,3}
BB{0,2}
BB?B?
B|BB|BBB
DBB GROUP (using the abstract `B{1,3}?' where B is any pattern)
S1
S2
S3
S{3}
SSS
S{3,3}
SNG GROUP (using the abstract `S{3}' where S is any pattern)
L1
L2
L3
A{2,}
AAA*
AA+
LWB GROUP (using the abstract `A{2,}' where A is any pattern)
C1
C2
C3
C4
C5
C6
[0-9a]
[0123456789a]
(0|1|2|3|4|5|6|7|8|9|a)
[^\x00-/:-`b-\177]
[\da]
(\d|a)
CCC GROUP (using the concrete example of `[0-9a]' and assuming an ASCII charset)
T1
T2
T3
T4
\a\$>
\x07\x24\x3E
\a[$]>
\007\036\062
LIT GROUP (using the concrete example `\a\$>' and assuming an ASCII charset)

Figure 1: Some possible refactorings

matched by the regex, and compose a string that is matched by the regex.

Our overall research questions are:

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

RQ2: Which refactorings have the strongest support based on matching strings?

RQ3: Which refactorings have the strongest support based on composing strings?

RQ4: What is the overlap in the refactoring suggested by RQ1, RQ2, and RQ3?

First we define a 'Functional Regex'(FR) as some regex that performs in a specific way. For many FRs, there are several concrete ways to express a single FR. We define a concrete regex(CR) as a regex expressed with a particular pattern String. Here is one illustration of these definitions:

TODO.NOW: create some examples for these terms

We identified 10 loose groups of FRs, described in this table:

TODO.NOW: create a table explaining the 10 groups

For each of these groups we created either two concrete versions of three FRs or three concrete versions of two FRs.

Each of the 10 categories had 6 concrete versions of some FR and so there are 60 CRs. For each CR, we selected 5 *example strings* designed to test the understanding of the CR. The idea is that different CRs may have different levels of readability, even when they are representing the same FR. We define readability as the ability to look at the CR and determine if an *example string* can be matched by it or not.

TODO.NOW: create some illustration of one matching subtask

Table 1: Matching metric example

String	`'RR*'`	Oracle	Response 1	Response 2
1	"ARROW"	✓	✓	✓
2	"qRs"	✓	✓	
3	"R0R"	✓	✓	✓
4	"qrs"		✓	
5	"98"			
	Score	1.00	0.80	0.80

4.1 Metrics

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

Matching: Given a regex and a set of strings, a participant determines which strings will be matched by the regex. The percentage of correct responses is the matching score. For example, consider regex `'RR*'` and five strings, which comes from our study, shown in Table 1, and the responses from two participants in the *Response 1* and *Response 2* columns. The oracle has the first three strings matching since they each contain at least one `R` character. *Response 1* answers correctly for the first three strings but incorrectly thinks the fourth string matches, so the matching score is $4/5 = 0.80$. *Response 2* misses the second string, so they also scored $4/5 = 0.80$.

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

1.

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

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

3.

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

4.

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

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

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

Figure 2: Participant Profiles, $n = 180$

TODO.NOW: describe how the string was tested against the regex, which libraries, etc.

4.2 Design

In Mechanical Turk, we designed a 180 tasks composed of 10 matching subtasks, so that each of the 60 CRs had 30 separate observations (each an average of 5 *example string* problems). These 1800 observations are what the analysis will focus on. The ordering of the regexes in each HIT was random to control for learning effects.

TODO.NOW: Were there qualification questions that required participants to demonstrate knowledge of regexes?

4.3 Participants

In total, there were 180 different participants in the study. A majority were male (83%) with an average age of 31. Most had at least an Associates degree (72%) and most were at least somewhat familiar with regexes prior to the study (87%). On average, participants compose 67 regexes per year with a range of 0 to 1000. Fittingly, participants read more regexes than they write with an average of 116 and a range from 0 to 2000. Figure 2 summarizes the self-reported participant characteristics from the qualification survey.

Basic regex knowledge test. 4/5 correct to pass.

5. RESULTS

(for rough draft...)

Looks like M6 and M8 are the best meta-refactorings according to ANOVA. If you peek at MTResults Processing.csv on google docs, M6 has the best refactoring...every OCTAL type should be converted to an OR or preferably a CCC.

M8 has a weak P value, but still ok in one case (0.12) and consistently says that 'aa*' should be written as 'a+'.

Looks like M0, M1, M2, M3 and M9 are very dependent on the regex chosen, so regex-specific refactorings like: 0.1401 `&d([aeiou][aeiou])z’` `&d([aeiou]{2})z’` 0.075 `[\t\r\f\n ]’` `[\s]’` 0.1024 `[a-f]([0-9]+)[a-f]’` `[a-f](\d+)[a-f]’` 0.1271 `[\{][\$](\d+[.]\d)[}]’` `\\\{\\\$(\d+\.\d)\}’` (from M0,M1,M2,M9 respectively)

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

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

Although one refactoring from M5 might be of slight interest: 0.1196 FALSE `tri[a-f]3’` `tri(a|b|c|d|e|f)3’`

TODO.MID: more data from the composition problems

6. DISCUSSION: REFACTORING OPPORTUNITIES

7. THREATS TO VALIDITY

7.1 Internal

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

7.2 External

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

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

8. CONCLUSION

Acknowledgements

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

Table 2: How frequently is each alternative expression style used?

code	R1	R2	R3	acc1	acc2	acc3	cmp1	cmp2	cmp2	Pacc	Pcmp
M0	`’^[1-9][0-9]*$’`	`’^[1-9][0-9]*$’`	NA	0.85	0.85	NA	27.0	26.3	NA	0.42	1.00
M1	`’^[1-9][0-9]*$’`	`’^[1-9][0-9]*$’`	NA	0.68	0.75	NA	21	23	NA	0.28	0.67
M2	`’^[1-9][0-9]*$’`	`’^[1-9][0-9]*$’`	NA	0.82	0.86	NA	23.3	25.3	NA	0.67	0.42
M3	`’^[1-9][0-9]*$’`	`’^[1-9][0-9]*$’`	`’^[1-9][0-9]*$’`	0.81	0.86	0.84	15.5	27.5	19.5	0.62	0.19

Table 3: How frequently is each alternative expression style used?

name	description	example	nPatterns	% patterns	nProjects	% projects
C1	CCC and RNG	`’^[1-9][0-9]*$’`	2,510	0.18	818	0.53
C2	CCC, no RNG or defaults	`’[aeiouy]’`	1,283	0.09	551	0.36
C3	OR of single chars	`’\\(t|n|r|"|\\)’`	156	0.01	174	0.11
C4	Contains NCCC	`’[^0-9]’`	1,935	0.14	776	0.50
C5	CCC and defaults, no RNG	`’[-+\d.]’`	675	0.05	382	0.25
C6	OR containing defaults	`’(\s|_)+’`	90	0.01	131	0.08
CCC REM	null	`’^\s*$’`	7,491	0.55	1,223	0.79
D1	DBB other than 0,1	`’^x{1,4}$’`	279	0.02	212	0.14
D2	DBB exactly 0,1	`’^[\n\r]{0,1}’`	23	0.00	25	0.02
D3	Contains QST	`’^https?://\S+$’`	1,871	0.14	646	0.42
D4	OR expressing repetition	`’^(Q|QQ)\<(.+)\>$’`	10	0.00	27	0.02
DBB REM	null	`’^[1-9][0-9]*$’`	11,491	0.85	1,473	0.95
LIT REM	null	`’(\w+)\s+(\d+)’`	319	0.02	538	0.35
T1	no HEX, OCT or CCC-wrapped	`’get_tag’`	12,482	0.92	1,485	0.96
T2	Contains HEX literal	`’(\x1b|~{)’`	479	0.04	243	0.16
T3	Contains CCC-wrapped	`’[$][{]\d+:([^}]+)[}]’`	307	0.02	268	0.17
T4	Contains OCT literal	`’[\041-\176]+:$’`	14	0.00	37	0.02
L1	LWB like M,, M>0	`’^_{2,}.*[^_]+_?$’`	91	0.01	166	0.11
L2	Contains KLE	`’^(.*?)’’’\s*(#.*)?$’`	6,017	0.44	1,097	0.71
L3	Contains ADD	`’[A-Z][a-z]+’`	6,003	0.44	1,207	0.78
LWB REM	null	`’-py([123]\.[0-9])$’`	3,979	0.29	944	0.61
OR REM	null	`’^[012TF\*]{9}$’`	11,495	0.85	1,489	0.96
R1	OR with prefix or suffix	`’^(def|class)\s+’`	1,529	0.11	603	0.39
R2	top-level OR	`’([ ]+_)|(_[ ]+)|([ ]+)’`	573	0.04	396	0.26
S1	Contains SNG	`’[^ ]*\.[A-Z]{3}’`	581	0.04	340	0.22
S2	Repeated non-literals	`’ff:ff:ff:ff:ff:ff’`	927	0.07	497	0.32
S3	DBB like M,M	`’U[\dA-F]{5,5}’`	27	0.00	32	0.02
SNG REM	null	`’@@BINARYDIR@@’`	12,209	0.90	1,498	0.97