Evaluation of Regular Expression Comprehension Tools

Andjey Ashwill
Purdue University
West Lafayette, Indiana, United States
aashwill@purdue.edu

Aiden Gonzalez
Purdue University
West Lafayette, Indiana, United States
gonza487@purdue.edu

Ethan Link
Purdue University
West Lafayette, Indiana, United States
link32@purdue.edu

Joshua Majors
Purdue University
West Lafayette, Indiana, United States
jmajors@purdue.edu

ABSTRACT

Introduction: Regular expressions are notoriously hard to intuitively comprehend. Debugging issues with them can also be difficult. Several techniques have been created to help alleviate this issue and make these expressions easier to understand.

State of Practice: Existing techniques include everything from diagrams to equivalent code to natural language descriptions. However, there is not a lot of empirical evidence to show which methods are most effective in aiding regular expression (regex) comprehension.

Contribution: In our work, we conduct an experiment to study and measure the effectiveness of several different comprehension techniques. By doing so, we are able to provide empirical support for the strongest of these techniques, encouraging its adoption to fight regex-related bugs in software.

Method: To make our contribution, we select three popular and distinct regex comprehension techniques and integrate them into an experimental tool. The tool has various measurement techniques to monitor user comprehension of regular expressions. After many trials with a comprehensive sample of developers, comprehension of the different techniques is analyzed, and findings are presented.

ACM Reference Format:
Andjey Ashwill, Aiden Gonzalez, Ethan Link, and Joshua Majors. 2022. Evaluation of Regular Expression Comprehension Tools. In *Proceedings of May 01,2022 (No Conference).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Regular expressions, or 'regexes' for short, are tools for programmers that allow for easy pattern matching in strings. Regexes can trace their roots to the 1940s and 50s where the first research was done on regular languages and finite automata [4]. In the 1960s regexes made the jump to the computing world when Ken Thompson, a developer of UNIX, implemented regexes as a search function in a text editor [17]. `grep`, the popular UNIX search tool, is an acronym for the search feature Thompson developed[1] [8].

Today, regexes are available in most programming languages via a standard library. As such, they have become a popular tool for developers. In 2018 researchers scanned Python Packages from PyPi, maintainers of Python Packages, and found over 63,000 *unique* regexes in about 72,750 packages [6]. Regexes are especially useful in security and reliability based contexts. Common use cases include sanitizing input, filtering spam, identifying harmful strings, and more. Given their importance and prevalence in the software that supports our everyday lives, it is concerning that they are simultaneously difficult to intuitively read, understand, debug, and verify [13]. While it is a known problem regexes are difficult to work with [13], little research has been conducted comparing the proposed solutions for regex comprehension. The purpose of this paper is to analyze existing regex comprehension and visualization techniques. The key contributions of this paper are as follows:

- Identify the strengths and weaknesses of existing regex comprehension techniques.
- Provide empirical evidence for which regex comprehension techniques are the most effective.
- Serve as an aid for the development of new regex comprehension methods to maximize their effectiveness.

To accomplish these goals, we selected three regular expression comprehension techniques and conducted an experiment with developers to determine which methods were the most effective. The rest of the paper is laid out as follows: Section 2 gives the relevant background of regexes and delves into issues with regex readability and proposed solutions, Section 3 discusses other work done in the area of regex readability, and Sections 4, 5, 6, and 7 discuss our experimental procedure and results. We conclude with discussion and potential future work (Section 8), followed by a recap of our findings (Section 9).

2 BACKGROUND

2.1 Regular Expressions

Regular expressions are a powerful tool to match patterns in strings. In its most basic form, occurrences of particular sub-strings will be matched by a regex consisting of just that sub-string. For example,

[1] The g in grep stands for global, re stands for regular expression, and the p stands for printing the search results.

No Conference, West Lafayette, IN,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

#	Regex	Fully Matched String(s)
1	aab	aab
2	[ab]b	ab or bb
3	[a-z]	a or f or z
4	[0-9A-F]	A or 7 or 9
5	\d	0 or 5 or 8
6	\D	d or E or ?
7	(aa)*b	b or aab or aaaab
8	\(\d\d\d\) \d\d\d-\d\d\d\d	(123) 456-7890
9	abc*	ab or abc or abccc

Table 1: Table of Regular Expressions and potential matching strings. While some matching strings are listed, there may be other strings that match that are *not* listed.

regex 1 in Table 1 could be used to simply match any occurrences of 'aab' in a string. String 1 in Table 1 is an example of a string which is 'fully matched' (meaning the regex matches the entire string, not just a part of it). Moving on from this, regex character classes (defined with square brackets) can be used to match any one of a range of characters. Regex 2 in Table 1 will fully match the strings 'ab' and 'bb'.

Ranges of characters can be used when defining character classes. For example, Regex 3 in Table 1 will match any lowercase letter. A regex to match a character in a hexadecimal string may look like Regex 4 in Table 1. Most languages have predefined character groups that act as shorthand for the bracket character class notation. In Python, Regex 5 in Table 1 can be used to match any digit ([0-9]) whereas Regex 6 in Table 1 is used to match any character that is *not* a digit.

Operators such as the Kleene star (*) are 'quantifiers'. The Kleene star in particular matches zero or more occurrences of the immediately preceding element. For example, Regex 7 in Table 1 could find matches such as 'aab' as well as 'aaaab' or even just 'b'.

If a programmer wanted to match a square bracket or parenthesis (as opposed to defining a character class or group) or any other control operator, they would simply escape the character using a '\', in most languages. Regex 8 in Table 1 could be used to match a US phone number with parenthesis around the area code (ex: '(123) 456-7890').

When regexes are compiled, they are often turned into a deterministic finite automaton (DFA). DFAs are finite state machines that transition between states upon input and have one or more "accept" states they can reach if given the right input sequence. For each new character inputted, the DFA transitions to its next state based on its predefined state transition graph (see example in figure 1). If there is no defined transition for the character, the input string is not considered a match and the DFA is reset to it's initial state. If the DFA is in an 'accept' state at the end of input, then a match has been found. The DFA pictured in figure 1 corresponds to Regex 9 in Table 1, with accept states denoted by a double outline.

If the DFA pictured in figure 1 were to test the string 'abcc', it would start in state S_0. The DFA would transition to S_1 upon input of the 'a' character. It would then receive characters 'b' and 'c' and transition through state S_2 then to S_3 as a result. After receiving the second 'c' character, the DFA would remain in the state S_3. Because

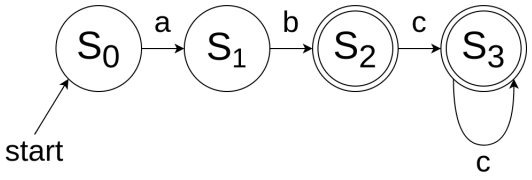

Figure 1: Example of DFA for Regex 9 in Table 1

there are no more input characters to consume and the DFA is in an accept state (S_3), the string is considered a match.

If we were to test the string 'afc' with this DFA, it would transition into S_1 after receiving the 'a', but because there are no defined transitions for the character 'f' from this state, the string would *not* be considered a match. Because DFA's have a visual representation, many regex visualization tools display a modified DFA to the user. Regex visualization tools are discussed in further detail in section 2.3.

2.2 Motivation - Regex Comprehension Issues

Anyone that has worked with regular expressions before can attest to its somewhat unconventional and dense syntax. There have been many studies demonstrating this issue empirically. [10, 13]. It has been found that regardless of an individual's background in regular expressions, longer expressions are much harder to read than shorter ones, and textual representations are much harder to understand than graphical representations [10]. Even when a developer finally reads and understands a regular expression, it has been shown that it is another milestone entirely to then validate and document them [13]. However, it doesn't have to be this way. Regular expression comprehension tools like the ones in this study can help developers read and understand regular expressions much faster than when they are presented without any assistance at all. A motivating example for this can be seen in figure 2. This begs the question: which of these comprehension tools is most effective, and does regex experience level affect this answer?

While previous studies have shown the effectiveness of visual explanations, none have directly compared multiple visual explanations, or even multiple visual explanations with another novel non-visual comprehension method. The primary motivation of this study then is to study which kinds of these advanced comprehension methods are most effective, and why. Going beyond this, the effectiveness of each method will be compared across experience levels to further study and compare the performance of several visual and non-visual comprehension methods to each other.

2.3 Regex Comprehension Solutions

Three comprehension methods have been selected to be a part of this study. While there are a myriad of options, we chose tools which represent three of the more popular categories of comprehension assistance:

- Explanation: Explains the functionality of each syntactic element of an expression, often with color-coding
- Visualization: Uses diagrams to visually demonstrate the functionality of a regular expression

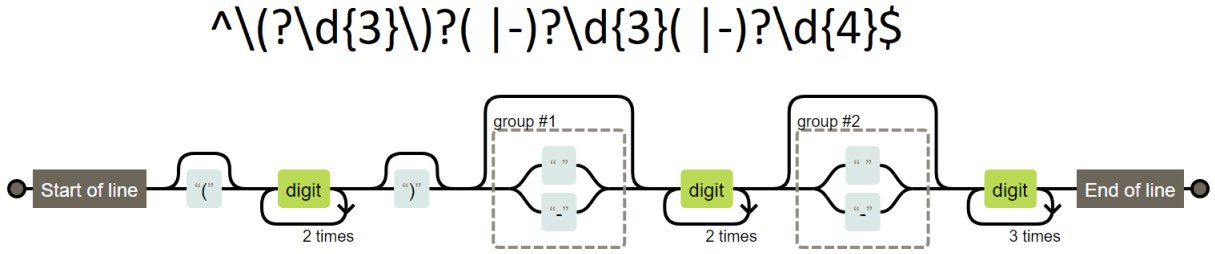

Figure 2: Phone number regular expression

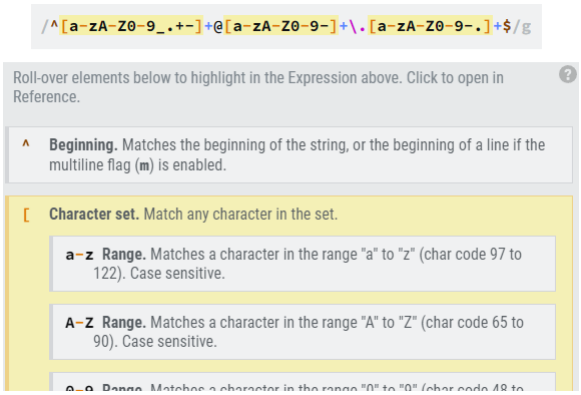

Figure 3: Example of RegExr output

- Generation: Accepts specifications, descriptions, or examples of accepted strings and generates a desired regular expression

One tool was then selected to represent each category.

2.3.1 Method 1: Explanation (RegExr). The first tool selected is RegExr, a color-coded explanation tool [15]. The tool works by highlighting the various syntactic structures within a regular expression, each with its own unique color, and explaining the meaning of the syntax (using the same color coding) below the expression itself. Figure 3 shows an example of RegExr output for a simple email-matching regular expression. RegExr was chosen specifically for its popularity, quality explanations, and open-source status. It will serve as a quality representative of the general 'visualization' and 'explanation' categories of comprehension tools.

2.3.2 Method 2: Visualization (RegExper). The second tool selected is RegExper, a simple yet effective visualization tool [1] that takes its inspiration from NFAs. The tool works by breaking down the structure of a regex and representing each major component as a separate state in a NFA diagram. Figure 4 shows an example of RegExper output (using the same input as for RegExr). Much like RegExr, RegExper was chosen specifically for its popularity, clear and intuitive diagrams, and open-source status. It will serve as a quality representative of the extensive 'NFA visualization' category of comprehension tools.

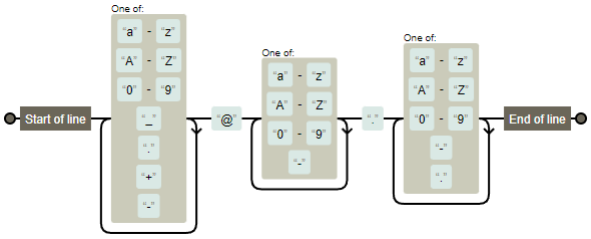

Figure 4: Example of RegExper output

2.3.3 Method 3: Generation (Grex). The third and final tool selected is 'Grex', an open-source regular expression generation tool [16]. The tool works by first accepting a list of strings that the generated regex should accept. Grex then builds a deterministic finite automaton (DFA) from the strings, reduces the DFA using a minimization algorithm, and then expresses the minimized DFA as a system of linear equations, which are then solved to produce the final regular expression which satisfies all DFA constraints. Figure 5 shows an example of output from this tool for a simple user ID regular expression. Grex was chosen for its unique approach to regular expression comprehension. Unlike the other two tools, Grex generates a completely new regular expression for the user instead of helping the user understand an existing regular expression. Because of this functionality, its output can very drastically based on the user-inputted strings and user-selected options (for the purposes of this study, participants were limited to default behavior). This gives it great potential for new and novel insights as it stands in stark contrast to more conventional tools. While it doesn't directly help developers understand an existing regex, by generating correct regular expressions for a set of example strings, Grex can help form the same correlations in reverse. It will serve as a representative for the "regular expression generation" style of comprehension tools.

3 RELATED WORKS

Several papers have been written studying developer's abilities to create, read, or validate regular expression strings [2, 5, 7, 9, 11–13]. For example, Michael, et al. [13] demonstrates a belief among a majority of surveyed developers that regexes are more complex than standard code and often daunting for beginners. They go on to show that even seasoned developers lack an awareness of the risks and vulnerabilities of regexes [13].

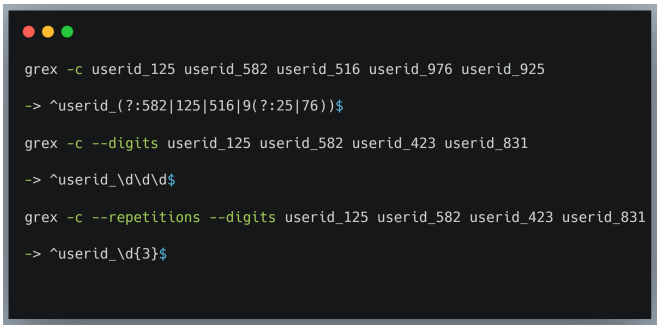

Figure 5: Example of Grex Input, Options, and Output

Many studies have demonstrated some of these hazards. Davis et al. have studied regular expression's portability across many languages that support them and the impacts of copying-and-pasting them from one to another in terms of both syntactic similarity and performance factors. Their study demonstrates an important realization that though many developers assume that the regex language is universal across programming languages, this is currently not true and there can be significant differences between the implementations of regular expressions in each studied language [7]. Bai et al. also encountered this phenomenon when studying the processes in which developers attempt to write and test regexes. This study found that often, at least as a starting point, developers will lead with a Google search followed by a copy-paste on the found result, but these results often led to compile errors [2]. They also found, when studying how Regular Expressions evolve over time and how they are tested, that 95% of regular expressions do not change from the time they are first implemented [18], and Stolee et al. found that that only 17% of regular expressions in the 1200+ GitHub projects studied had any associated test input [19]. These statistics are likely products of developers' struggles to understand regular expressions, leading to an unwillingness to implement test cases or iterate to improve regular expressions already located in a code base.

There are also approaches taken by previous research groups to help increase developers' ability to work with regular expressions through validation and/or visualization. Larson describes a validation tool, "Automatic Checking of Regular Expressions," or ACRE, created to check regular expression patterns for common mistakes in syntax and format [11]. Tools like this, along with some compilers or even IDEs such as Visual Studio, can assist developers in identifying issues with their regexes before testing or deploying them. However, these tools come with the implicit drawback that they are only able to identify the issues they are explicitly designed to identify, which can itself become a challenging task given the discussed complexity of regex strings. There is a different approach taken by Larson and Kirk in their earlier paper, in which they describe a tool called EGRET (Evil Generation of Regular Expression Tests). This tool takes a regular expression as input and develops test strings which this regular expression can be tested against, with the intention of purposefully creating strings that demonstrate potential issues with the previously input regex. The generated strings are tested against the regex and the results are displayed to the user, allowing the user themselves to have a clearer idea of how their pattern is functioning, and potentially give insight into unintended bugs [12]. Blackwell describes "Programming By Example" [3], an algorithm that takes in one or more words and creates a regular expression that fits the selected words. The program then highlights all the words in the available text that fit the expression generated by the SWYN algorithm [3]. Tools like the ones listed here have been shown, especially in the case of novice use, to lower the barrier for entry when working with regular expressions.

Research has been done investigating syntactical or representative strategies to improve software developer's abilities to work with regular expressions. Chapman et al. analyzed and conducted community surveys to determine which features and styles of representations within the regex language were determined to be the most understandable among developers [5]. This work studied the symbols and representations present within the confines of the regex language, whereas the goal of our work is to expand the scope into comprehension strategies located outside of the defined regex syntax context. More directly related to our proposed work are papers that discuss strategies to improve the readability of regexes. Previously mentioned, Erwig et al. developed strategies for translating complex-appearing regex patterns into a format that either more closely represents the string the user is attempting to match, or a format that can be broken down into its more readily understandable pieces [9].

Finally, our approach is not to create our own comprehension strategy as demonstrated by Erwig et al. [9], but rather to evaluate visualization or comprehension strategies already created to determine their actual effectiveness in terms of developer understanding and usability. Similar studies have been done by both Blackwell [3] and by Hollmann et al. [10] in which they studied the differences in readability between a purely textual representation and a more graphical representation of the same expressions. In Blackwell's SWYN paper[3], he also describes an experiment he ran to test different regular expression display formats and a human's ability to use the information to solve questions. Blackwell [3] found that graphical representations, specifically procedural graphics, greatly reduce time taken to answer questions as well as the incorrect answer rates when compared to the expression itself as well as textual representations. Hollmann's [10] experiment also found that using a graphical representation improved the readability of regular expressions significantly, resulting in subjects answering in less than 50% of the time compared to the textual representation, with even greater improvements for medium length (28 characters) and long (38 characters) regular expressions. Our study hopes to further explore these findings by directly comparing the effectiveness of two different kinds of graphical representations and experimenting with a regular expression generator [16].

4 RESEARCH QUESTIONS

The following questions will be answered using the data collected during this study:

- **RQ1:** Which comprehension tool caused the greatest improvement in understanding over baseline?
- **RQ2:** Which comprehension tool helped beginners more? Which best assisted more experienced participants?

- **RQ3:** Which comprehension tool had the biggest impact with the least use?

Answering these questions will allow us to understand and compare the scope of each comprehension tool's impact with the others.

4.1 Hypotheses

In regards to research question 1, we expected RegExper to be the most effective tool in increasing participant's comprehension over baseline. We believed this as a result of studies done by Blackwell and Hollmann, which both previously demonstrated that graphical representations showed significant increases in regex correctness [10] [3].

For Research question 2, we also believed that RegExper would similarly benefit beginners the most, as the simplicity and innate ability for diagrams to convey information would be most beneficial to those lacking in significant regular expression experience. On the other hand, for Advanced participants we believed that RegExr would be the most beneficial, as it allows more options and more forms of information that we think Advanced users would be able to leverage to their advantage. Blackwell shows in his research, citing that in the case of novices when verbose representations were encountered early in the experiment performance declined, whereas graphical representations, especially when presented early accounted for the largest improvement in both accuracy and time-to-respond [3].

Finally, for research question 3, we believed RegExper again would have the most impact with the least usage. Though Grex is seemingly simple to use, it can often require significant amounts of input to get the desired output. Also, as mentioned above, RegExr provides more freedom and power to the users in how they are willing to test, however the amount of information output by this tool could increase complexity and confusion. We think the simplest tool with the single most significant output is RegExper, and that's why we hypothesized that it would have the biggest impact per use.

5 SURVEY TOOL DESIGN

To study the effectiveness of various regex comprehension tools and answer our research questions, our experiment utilizes a custom experimental tool consisting of a front-end survey for data collection and back-end server for regex processing and data logging. More specifically, the architecture of the experimental tool used in this study can be found in Figure 6. Participants from the pool of Software Engineering students at Purdue University self-selected to participate by volunteering to take our survey in exchange for a meal and a chance to win one of five $10 Amazon gift cards (via lottery of participants). The study was conducted within-subjects, with each subject using each of the three Regex comprehension tools in a single experiment run.

5.1 The Pilot Study

We first collect background information about the participant's familiarity with regexes. We ask a total of 4 questions:

(1) Self-rating of level of familiarity with regular expressions (Beginner, Intermediate, or Advanced)

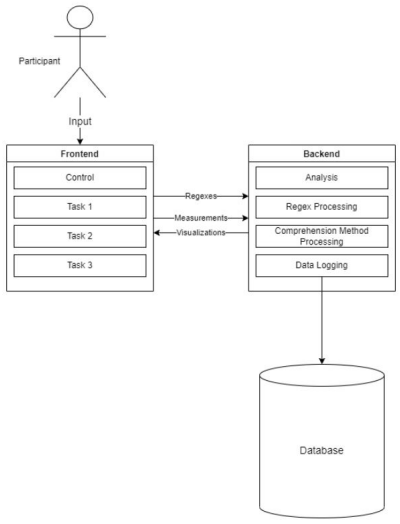

Figure 6: Architecture of Experimental Tool

(2) When they last worked with a regular expression (within the past week, 3 months, year, etc)
(3) How many unique regular expressions they have ever worked with (0, 1-5, 6-15, etc)
(4) How long ago did they first encounter regular expressions (Never, less than 3 years, 3-5 years, etc)

For question 1, we ask participants to self rate their familiarity. However, we provide guidelines to help reduce the subjectivity of participant responses. The criteria for beginner, intermediate, and advanced are as follows:

(1) Beginner: Have basic experience using only characters and simple quantifiers (such as `/(abd)*tf/`)
(2) Intermediate: Have used more sophisticated features like non-greedy quantifiers (`/a+?/`) and character classes (`/\d \w [abc] [^\d]/`)
(3) Advanced: Have used more advanced features like exact quantifiers (`/abc{4}(def){2,6}/`) and escaping characters (`/\$.co/`)

These four questions are intended to give us a general sense of a participant's experience with regexes. These questions were selected by the team after a discussion of which questions would be best to estimate experience with regex. Further research could be done to improve the selection of these questions.

The regular expression challenge portion then begins. This portion consists of four sections: one for control (no assistive tools) and one for each regular expression comprehension method. The control section helps contextualize the findings from the tool sections and provide a baseline for varying level of prior experience and skill with regular expressions between participants. Each of these 4 sections consists of three unique challenges:

(1) Write matching string
(2) Create a regex
(3) Modify a regex

Examples of each of these challenges can be viewed in Table 2 and a comprehensive list of challenges can be found in Appendix A.

These challenges were specifically selected to give insights into Regular Expression comprehension beyond simply 'can you read this regex.' Participants will not only need to comprehend a regular expression, but they will need to understand how it works in order to alter its function. This helps to expose the strengths and weaknesses of each regex comprehension technique. Our team attempted to develop the questions used in this research in such a way that the difficulty is similar and a variety of regex functions and syntax are in play.

In a further effort to control for prior regular expression knowledge, a regular expression reference sheet was provided to participants in a separate window. This one-page document contained a breakdown of basic regular expression constructs and syntax. In this way, users with less syntactical knowledge were given a reference and not limited in their participation by their previous knowledge and experience. The number of times the participant referenced the reference sheet could also be used as an indicator of prior knowledge, but also as another data point to observe how participants utilized the sheet as compared to our selected comprehension tools.

When deciding on the length of the survey, we evaluated multiple factors such as number of questions per tool, within-subjects or between-subjects, and the difficulty of questions. Our goal was to find a ideal balance between the amount of questions that gave us enough data per participant and not so many questions as to overwhelm participants. As shown above, we landed on 3 questions section, leaving 12 questions total (excluding experience questions). In order to ensure the survey isn't taking too much time for the participants, we decided to limit time to 2 minutes and 30 seconds per question. Past this time, the survey allowed users to advance to the next question regardless of the correctness of their answer.

5.2 Backend Data

Throughout the experiment, logged data was sent to the back-end to be stored in a database for later analysis. The data from each participant was tied to a unique participant ID in order to protect the privacy of each participant in alignment with our IRB protocol. Data in the back-end is stored in tables according to the database diagram in Figure 7. For each challenge, the number of attempts at submission was recorded as "numAttempts", the time taken to finish will be recorded as "timeToComplete", and a flag to track whether or not the participant answered the question correctly was stored in the database as "correct."

6 METHODOLOGY

6.1 Survey Details

As our research deals with complex human elements, it is important that we control for any variables that might bias the results of our study through careful experimental design.

One important factor in our survey design is creating challenges of comparable difficulty. Because we want to measure the effectiveness of the regex tools, we need to ask the participant different regex challenges for each tool. Furthermore, we avoid re-using the same features of the regex language in subsequent challenges

Figure 7: Database Diagram

to minimize the probability that a user might become more comfortable with a language feature in a way that would artificially enhance their performance when using later tools in the survey.

These requirements make it difficult to ensure that the regex challenges created are not only unique, but also comparable in difficulty. We sought to create questions that were, in our judgement, comparable in difficulty. However, there was no real quantitative way for us to objectively assess the difficulty of the challenges we created. Given this limitation, after manually creating the questions that we felt were comparable in difficulty, we also developed a simple balancing algorithm to help ensure that each question was paired with each tool an even number of times.

To do this, the balancing algorithm first finds the number of times each challenge has been assigned to each tool over all previous survey trials. It then assigns each challenge to the tool that has seen that challenge the least. This ensures that every participant is asked the same 12 challenges in the end (just in a different order) and each challenge has roughly equal pairings with each tool. This helps to mitigate the impact of challenge difficulty variation on comparative tool performance, because the impact of one potentially difficult challenge would be fairly distributed amongst the tools. As a result of this balancing algorithm, across 12 survey runs each of the 12 challenge questions was assigned to each survey section (Control, RegExr, RegExper, and Grex) 3 times. For a more visual explanation of this even final distribution, see table 4 in Appendix A.

As an example, "Create D" was found at the completion of the study to have been the most difficult question on the survey, as evidenced by its lower success rate and longer completion times. This is something that is difficult to predict ahead of time. However, because we implementing our balancing algorithm, the effect of this hard problem was evenly and fairly distributed amongst all of

Challenge Type	Question	Potential Answer
Write a matching string	... that fits this regular expression: `^[A-Z]{3,5}[0-9]*$`	AAA2
Create a regex	... that matches anything comprised of numbers and underscores	`^[0-9_]+$`
Modify a regex	... accept both '.com' and '.edu': `^[A-Za-z0-9_]+@[\w]+\.com$`	`^[A-Za-z0-9_]+@[\w]+\.(com|edu)$`

Table 2: Example challenges participants had to complete with potential answers. A comprehensive list of challenges can be found in section 10.2

the tools (and the control section), preventing it from affecting the quality of our results.

There are specific data points that we wanted our experiment to capture that could not be automatically tracked by our survey. Since this research was designed to evaluate Regex Comprehension tools, we want to gather qualitative feedback about the tools as well as the survey in general from our participants. Following the completion of the survey, we ask the participants three questions:

(1) Which tool was your favorite?
(2) Which tool was your least favorite?
(3) Any other thoughts about the tools or survey?

This qualitative feedback gives us access information that cannot be recorded through the survey. The first two questions about user preference tell us which tools participants felt contributed most to their success and which tools users preferred. Certain tools may score poorer on understanding metrics but may still help users overall. For example, we assume when a user takes longer on a challenge, they show less understanding. This assumption may be incorrect because participants may be spending extra time using the tool but in fact understand the regex in the challenge to a higher degree. Theories such as this would be impossible to gain insight into without asking users these questions. We also wanted to understand which tools user *enjoyed* using the most. If developers are given a tool that is difficult and unpleasant to use, they will most likely never opt to use this tool when developing with regexes, no matter how effective it may be.

Logistically, metrics such as the time elapsed and number of attempts taken to correctly answer a challenge are tracked programmatically by the survey. The screen is recorded during the survey to capture the number of interactions with each tool and the number of interactions with the cheat sheet, as these could not be captured programmatically. We define an interaction with a tool as an event where the user inputs new text into a tool, and an interaction with the cheat sheet as an event where the user flips to the separate browser tab containing the sheet.

After our survey has concluded, we manually extract tool and cheat sheet interactions from the screen recording and collate it with data recorded through the survey database.

6.2 Calculated Metrics

After designing our survey and collecting data, we develop a methodology to understand the data, specifically seeking to develop metrics that help us understand how effective a regex tool is at helping a participant interact with a regex.

Among the data we collected, we have three measurements we can use to help us accomplish this: how long a participant spent on a challenge, how many attempts they made on a challenge, and whether or not they correctly answered the challenge before time elapsed (correctness).

We developed a metric that uses these three quantitative measurements to provide an estimate of understanding. We argue generally that lower time taken to correctly answer a regex challenge means the participant has been more effective in their interactions with that regex than if the same participant had taken longer. We argue that less attempts taken to correctly answer a question indicates that the participant has been more effective in their interactions with that regex than if the same participant had required more attempts. Finally, if a participant has not been able to correctly answer a regex challenge before time expires, we argue that understanding should be set to zero, as we are unable to provide a more accurate assessment of their effectiveness at interacting with the regex, and have also observed that they were ultimately ineffective at answering the challenge.

$$understanding = correctness \cdot \left(\frac{1}{time} + \frac{1}{attempts}\right) \cdot 0.5 \quad (1)$$

The ranges for the values are as follows: $correctness \in \{0, 1\}$, $time \in [1 : 150]$, and $attempts \in [1, \infty)$

Equation 1 mathematically expresses these interpretations to translate our quantitative experimental measurements to qualitative numerical values that estimate understanding, or how effectively a user was able to interact with a regex.

We make generalized assumptions regarding linearity (between time elapsed, the number of attempts taken and understanding) and assume equal importance for time elapsed and understanding. While reality is more nuanced than these estimations describe, these assumptions simplify the model in absence of further data or studies to better parameterize it. Correctness is used as a indicator function to set understanding to zero if a participant did not correctly solve the challenge in time. We take the inverse of time elapsed and number of attempts required to linearly express that less time and fewer attempts result in increased 'understanding', and we add them together and divide by two to create a final value for understanding that will be at least zero and at most one.

We also want to assess if there is a difference in which tools were helpful depending on how experienced a given participant is. Similar to the understanding metric, we have no way to directly measure participant experience, but we ask the user several regex experience related questions at the start of the survey, and we use this information to create an *experience* estimate. (These questions and supporting rational are described in more detail in section 5.)

To create this experience metric, we map multiple choice survey responses to integer scores, where larger numbers indicate greater experience. We believe that more recent work with regexes should also translate to greater familiarity with regexes than less recent

work. We also believe that the more unique regexes an individual works with indicates greater familiarity, and that a longer history of working with regexes also indicates greater familiarity. Further rationale for the selection of these questions can be found in 5. We create a mapping between each survey response and an integer. This mapping can be viewed in Table 3, where higher numbers indicate greater experience with regexes.

Finally we use these integer values to create a new metric for participant experience that is computed for each participant. This metric is described in equation 2. The variables in equation 2 correspond to rows in table 3.

$$exp = (skill) + (last\ worked) + (unique) + (first\ time) \quad (2)$$

We weight each response (skill, first worked with regex, number of unique regexes and most recent experience with regex) equally and compute a value for *experience* that is the sum of these responses. As with the experience equation, further research would improve the estimation provided by the metric, but this rough equation captures the general principle that positive increases in participant interactions with regexes should yield a greater value for the *experience* metric.

After computing the experience metric for all participants, we normalize experience values for all participants between 0 and 1 and divide along the mean to partition participants into less experienced/more experienced groups. Participants with *experience* beneath the mean were considered less experienced and those at or above the mean we consider more experienced.

6.3 Answering Research Questions

In section 4, research questions were laid out and hypotheses were stated. These hypotheses will be evaluated as follows:

- **RQ/Hypothesis 1:** Improvement in the understanding metric over baseline (with additional consideration given to qualitative feedback) will be used to determine the most effective tool.
- **RQ/Hypothesis 2:** Improvement in the understanding metric over baseline for beginners vs advanced users (sorted by the experience metric) will be used to determine which tool was most effective for each group.
- **RQ/Hypothesis 3:** Change in the understanding metric vs number of tool uses (with additional consideration given to number of 'cheat sheet' uses and qualitative feedback) will be used to determine which tool was the most efficient.

7 RESULTS

Quantitative results directly from the survey will now be presented, followed by qualitative results derived from the post-survey interviews conducted with participants. Trends and discrepancies will then be discussed in section 8.

7.1 Quantitative Data

The first metric we will look at is the time to complete a task when using different comprehension methods. This gives us an idea of how *efficient* a user is when using a specific comprehension method for different tasks. A breakdown of the time it takes the participants

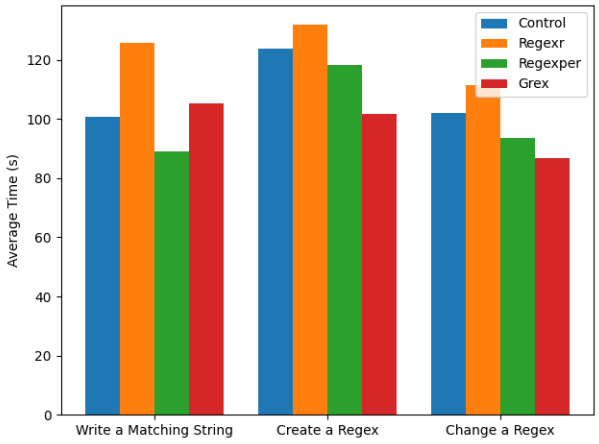

Figure 8: Average time taken for each comprehension method broken down by task type.

to complete the task using different regex tools can be seen in figure 8. As seen in the figure, Grex appeared to be the best tool for changing a regex as well as creating a new regex, followed closely by RegExper (although RegExper did see the shortest average time for users to create a matching string).

The next metric gives us insight into the users *preference* of regex comprehension method. Figure 9 shows the average number of interactions a user had with a tool while performing a given task. It appears that Grex was the least popular of the 3 tools tested. It is important to note that when asked to change a regex, the average number of interactions with Grex was less than 1, meaning in many cases some participants did not use Grex at all when answering the question. RegExper was used on average 1 time. This makes sense because RegExper is essentially a NFA viewer, and the user only needs to view the NFA for a single regex. Because we defined user interaction as any new input to the tool and RegExr has a method for checking text for matches in a Regex, it saw a spike for the task of writing a matching string. Many users would test multiple inputs before finding and submitting their answer.

During the survey, participants were given a reference sheet for regex operators. This reference sheet can be viewed in Appendix A. We measured the number of times a participant switched tabs to view this reference sheet and plotted it against the number of interactions a participant had with a given tool. These plots can be seen in figure 10.

We found that RegExper and RegExr both have flatter correlations than Grex. RegExr has a slightly negative slope. This indicates that as users use the tool more often, they rely less on the reference sheet. This makes sense as RegExr is able to provide much of the same information about regex operators that can be found on the reference sheet. Meanwhile, RegExper has a positive slope. This indicates that as the problem scales, participants use the reference sheet in *conjunction* with RegExper.

Grex's steep slope in figure 10 indicates that as problem difficulty increased, participants looked at the reference sheet more and increasingly abandoned use of Grex. To this end, many data points

		XP Points For Answer				
Variable Name	Questions	0	1	2	3	4
skill	How do you rate your familiarity/skill with regexes?	-	Beginner	Intermediate	Advanced	-
last worked	When was the last time you worked with a regex?	-	< 12 Months	< 3 Months	< 1 Week	-
unique	How many unique regex have you worked with?	0	1-5	6-15	16-30	30-75
first time	How long ago did you first encounter a regex?	Never	Less than 3 years	3-5 years	-	-

Table 3: Table detailing experience points distribution

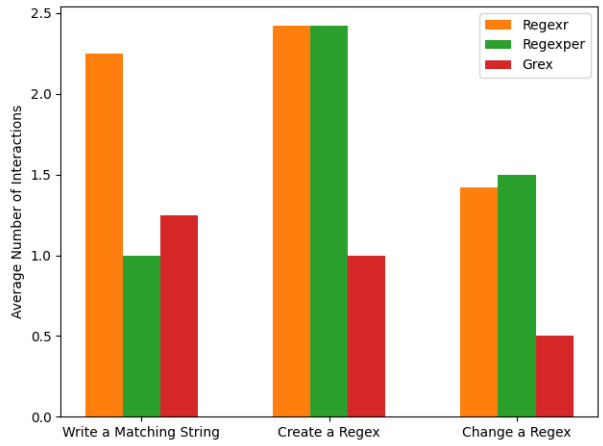

Figure 9: Average number of interactions for each comprehension method broken down by task type.

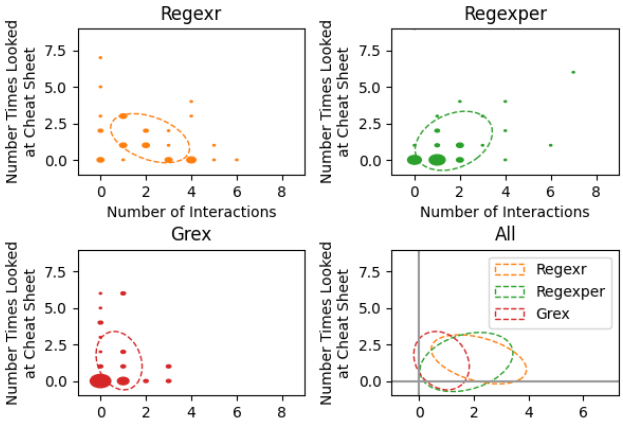

Figure 10: Correlation between number of times using cheat sheet while using each tool

show 0 interactions with Grex and several interactions with the reference sheet. This indicates many users did not even attempt to use Grex to solve their problem. This poses the question: why do we see such an improvement with Grex if participants are not using Grex? Discussion of this question can be found in section 8.

Next we looked at how understanding for each participant changes while using different regex comprehension methods. We use our understanding metric discussed in section 6 (equation 1). The results from each individual participant broken down by comprehension tool can be viewed in figure 11. Participants are sorted by their experience with the most experienced participant listed first and least experienced participant listed last. Participant experience is determined by the experience metric also discussed in section 6 (equation 2).

However it can be difficult to view major trends when viewing data for each participant. For this reason, we group participants by their experience into two classes: advanced and beginner using the experience metric. We sorted participants using this metric and put the upper half of the participants in 'advanced' and the lower half in 'beginner' (roughly). We then averaged understanding by tool type and plotted in figure 12. From figure 12, we can easily see that advanced developers found RegExper very useful, and RegExr and Grex about as useful as just a reference sheet. It is also noteworthy that RegExr was not only unhelpful to beginners, but was actually detrimental to their performance. RegExper and Grex scored similarly for beginners, with a slight edge going to Grex (but again, this result is scrutinized in section 8).

7.2 Qualitative Data

Next, we discuss the results from participant interviews. Nearly every participant (92% of participants) claimed RegExper is their favorite comprehension tool. Only 8% (1 advanced participant) selected Grex as their favorite comprehension tool and no participants selected RegExr as their favorite. People said they enjoyed using RegExper because of the visualization. They are able to work their way through graph displayed on RegExper to figure out what the regex is representing. Many participants (especially those who are less experienced) felt RegExper gives a nice introduction to regexes, so much so that they are able to learn from it and apply these learned concepts even when RegExper is not available.

The same 92% of participants that said RegExper is their favorite said Grex is their least favorite comprehension method of the 3 they used. Many people felt that it did not work and some participants going as far to say that Grex is 'basically useless.' Some users commented that RegExr may be nice if they had more time to experiment and learn how to use it. While Grex seems to be hated by most participants, it quantitatively seemed to help users' understanding more than any other tool, especially among beginner users. In the next section, 8, we discuss potential reasons for these discrepancies.

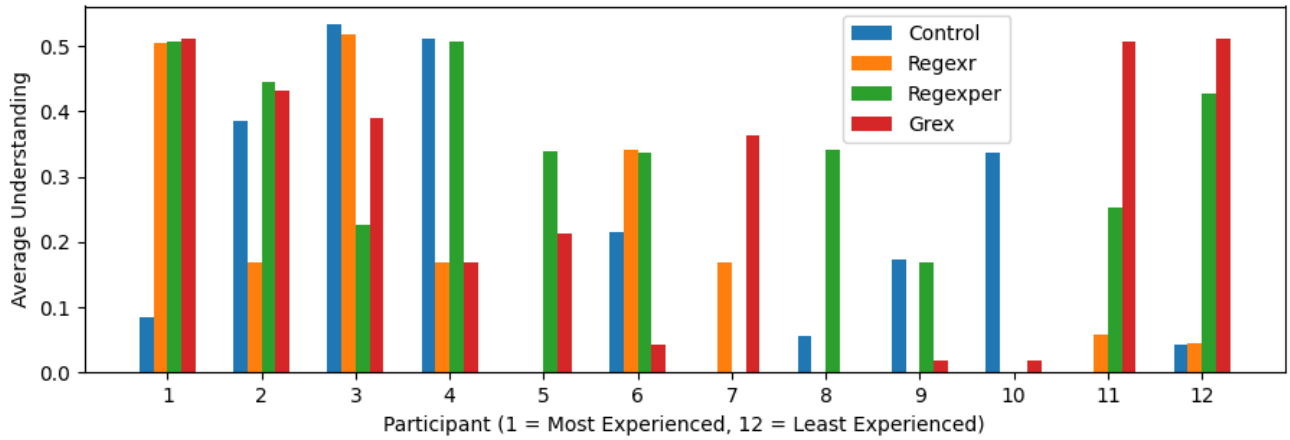

Figure 11: Average understanding metric for each participant.

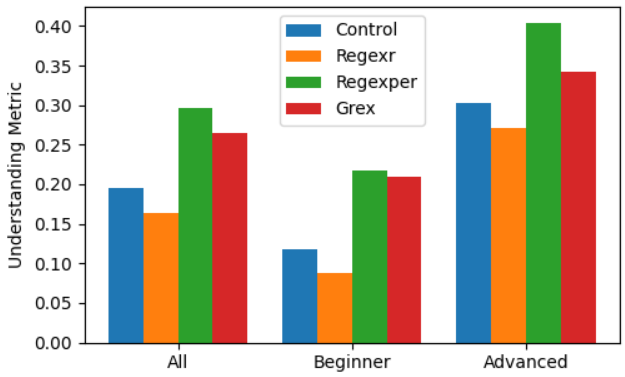

Figure 12: Average understanding metric broken down by participant experience level.

8 DISCUSSION

When comparing the results to our research questions, RegExper (the NFA tool) appears to be the answer to all three. It displayed the biggest immediate improvement in understanding over control data, helped both beginners and experienced users the most, and had the biggest impact on comprehension with the least amount of use required (other than Grex, but observations suggest this is a faulty finding).

We believe that RegExper is the most effective because a simple graphical representation of a regular expression is often the quickest and easiest way to explain the functionality of a regular expression, which aligns with the performance improvements seen with graphical representations in Blackwell's initial study [3]. It succinctly and accurately conveys the information graphically that is difficult to capture in text. RegExper also abstracts regex operators into control flow arrows. For example, a Kleene Star (*) is represented as a looping arrow. This leads to, especially in the case of beginners, a greatly increased explicit understanding on an expression's function and how to use it.

However, RegExr also showed strong performance, especially among experienced users. Those that were highly knowledgeable in regular expressions immediately knew how to take advantage of RegExr's explanation and testing features to quickly break down a regular expression and test a variety of input strings. RegExr is a tool that was effective for every kind of question, if used properly. Beginners were a bit overwhelmed by its complexity and more detailed analysis when compared to RegExper's simple NFA diagram.

As previously mentioned, over the course of the survey the participants likely learn more about regular expressions. As a result, participants have a stronger understanding and become more comfortable interacting with them in later questions. While we automatically balanced the order of the questions to try to combat varying difficulty, we did not practice the same technique with the tools themselves: RegExr was always first, RegExper was always second, and Grex was always third. Though Grex was the most disliked according to our participants and least used (even compared to the cheat sheet), it shows the greatest performance improvement, especially for regular expression beginners, when compared to other tools and the control questions. We do not believe that this is due to Grex being a superior tool, but rather an increased level of skill, comfort, and familiarity that participants gained with regular expressions as the survey went on. In other words, we believe the beginner-level participants "warmed up" over time, and that accounts for the improvements in performance seen with Grex. Meanwhile, more advanced users (as identified with our experience metric) performed better with RegExper, as they exhibited little to no learning over the course of the survey and thus did not inflate Grex's performance. This behavior is clearly shown in figure 12. Looking at the beginner group in figure 12, we can see they start out with low understanding, but when the beginner participants encounter RegExper, they see a huge jump in understanding and are able to apply their newfound understanding during the Grex phase of the survey. This behavior is consistent with what some participants explained during their interview after taking the survey.

Another factor that needs to be considered when looking at the results is the effect imposing a time-limit has on the participant. Powers and Fowles conducted a GRE writing exam with prospective

graduate students, and gathered data showing that over 60% of self-reported average-speed test-takers report feeling at least somewhat under pressure during a timed writing test [14]. This study also demonstrated that scores significantly decreased in the shorter time limit used in this test, affecting slower test-takers more than faster ones [14]. The overarching pressure a time limit can impose on a test-taker cannot be ignored, especially for beginner-level participants or slower test takers.

It was mentioned multiple times in our post-survey interviews with participants that they sometimes felt that they didn't have time to learn how to use the tools. One of the advanced participants mentioned that the RegExr tool looked extensive and like it would be the most powerful, but they never actually got the chance to use it. It's hard to say exactly what effect the time-limit might show in the data, though our theory is that more complex tools or tools with more options would become less effective and/or less interacted with in our study. RegExr, in our team's judgement, is the most visually complicated tool, and also offers many different tabs and options to give the user the full power of the tool, however it is possible that in this survey this extra complexity paired with a short time-per-question potentially decreased the effectiveness this tool otherwise would have demonstrated.

8.1 Potential Improvements

There are a few ways we can see that the learning phenomenon could be avoided if this study were to be repeated. One way would be to add some sort of basic tutorial that instructs beginners and experienced users alike on the use of the survey before the survey starts. This would help users understand how to use the surveys and especially the regular expression tools ahead of time, so the time to learn to use the tool is not a factor in determining its comprehension performance. Additionally, balancing the order of the tools like we balanced the order of the questions would help distribute the "learning effect" among all the tools and thus greatly reduce its impact on the final results.

Participants also took very different approaches to the problems in a way that may have affected results. Some submitted their answers as many times as possible knowing that the survey would validate their answer for them until they reached the time limit. Other participants only submitted their final answer once, spending the entire allotted time for a question perfecting their answer. Additionally, as participants learned that they could in fact submit as many times as they like, they submitted more on later questions than earlier questions, further skewing their performance across different sections. This greatly affected 'time to complete' and 'number of attempts' from one participant to the next. While we could compare the overall performance of the tools, we could never truly compare one participant to another. Restructuring the survey a bit so that participants are more aware that they can check their answers as many times as they like from the beginning could help reduce this variance and improve results.

To make Grex more useful in this study, options customizing its output could have been made available to users in the frontend survey. A demonstration of such a case can be seen in figure 5. The first command represents the output offered to participants in our study. The resulting regex covers all input strings, but no more. Instead, if users were allowed to access the `--digits` and `--repetitions` options, they could receive output that is useful and would have led them to the correct answer for many of the regex challenges given. This extra options that were not given to the users would surely make Grex more useful and thus change the survey results.

Finally, while we tracked how many times users interacted with a tool, we didn't track the length of each interaction or how much time they used a tool overall. Measuring and analyzing this statistic might help explain differences in performance better, or expose an entirely new insight that had previously been overlooked.

9 CONCLUSION

In conclusion, using our data we were able to make the following determinations regarding our research questions:

- RQ1: Overall, RegExper was most effective for improving performance over baseline.
- RQ2: While RegExper clearly helped both experienced and inexperienced users, RegExr was more useful and effective for experienced users.
- RQ3: Grex data looked good because it was used less and participants learned over the course of the study, but in reality RegExper seemed to aid in understanding the most while requiring the least interaction.

RegExper's strong performance serves as yet another experimental demonstration of the superiority of graphical comprehension methods and the effectiveness of tools that leverage them. Its also no surprise that RegExper was the best tool for beginners, as its visual explanations can effectively communicate the behavior of regular expressions to those that may not know much about how they work. Meanwhile, RegExr's more advanced syntax-highlighting, explanation, and string testing functionality makes it a better fit for more advanced users who are looking for more than a simple visualization. Grex on the other hand, while liked by some advanced users, was found to be less useful as a comprehension tool by most participants due to its less conventional approach and limited default output behavior.

REFERENCES

[1] Jeff Avallone. Regexper: Regular expression visualization site. https://regexper.com/, September 2020.

[2] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 197–208. IEEE Press, 2019.

[3] Alan F. Blackwell. Chapter 13 - swyn: A visual representation for regular expressions. In Henry Lieberman, editor, *Your Wish is My Command*, Interactive Technologies, pages 245–XIII. Morgan Kaufmann, San Francisco, 2001.

[4] Tim Carmody. A history of regular expressions and artificial intelligence. https://kottke.org/21/07/a-history-of-regular-expressions-and-artificial-intelligence.

[5] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 405–416, 2017.

[6] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[7] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the*

2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 443–454, New York, NY, USA, 2019. Association for Computing Machinery.

[8] David Delony. How grep got its name? the history behind grep's creation. /urlhttps://www.makeuseof.com/how-grep-got-its-name-the-history-behind-greps-creation.

[9] Martin Erwig and Rahul Gopinath. Explanations for regular expressions. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, page 394–408, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Niklas Hollmann and Stefan Hanenberg. An empirical study on the readability of regular expressions: Textual versus graphical. In *Proceedings of the 2017 IEEE Working Conference on Software Visualization (VISSOFT)*, IEEE 2017, pages 74–84, Shanghai, China, 2017. IEEE.

[11] Eric Larson. Automatic checking of regular expressions. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 225–234, 2018.

[12] Eric Larson and Anna Kirk. Generating evil test strings for regular expressions. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 309–319, 2016.

[13] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 415–426, 2019.

[14] Donald E. Powers and Mary E. Fowles. Effects of applying different time limits to a proposed gre writing test. /urlhttps://onlinelibrary.wiley.com/doi/abs/10.1002/j.2333-8504.1996.tb01706.x, 1996.

[15] Grant Skinner. Regexr: Learn, build, and test regex. https://regexr.com/, July 2021.

[16] Peter M. Stahl. Grex: A command-line tool and library for generating regular expressions from user-provided test cases. https://github.com/pemistahl/grex, April 2022.

[17] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[18] Peipei Wang, Gina R. Bai, and Kathryn T. Stolee. Exploring regular expression evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–513, 2019.

[19] Peipei Wang and Kathryn T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 668–678, New York, NY, USA, 2018. Association for Computing Machinery.

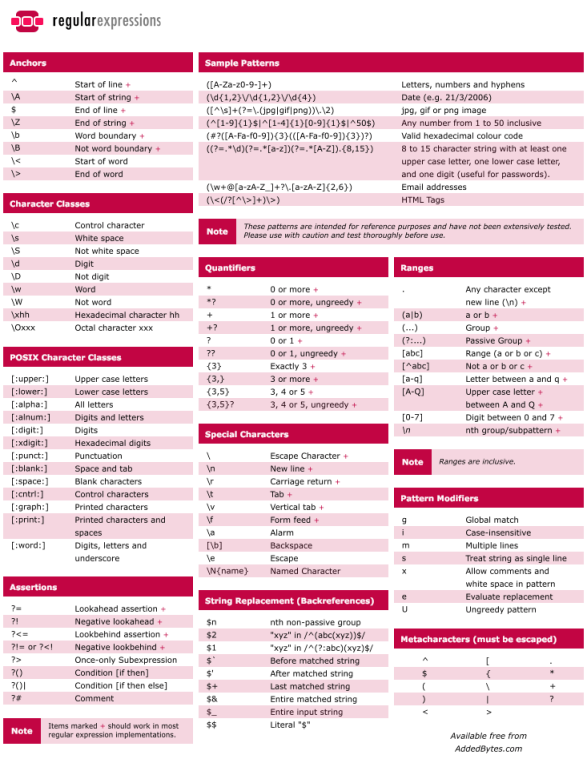

Figure 13: Reference Sheet given to participants. Link to larger image can be found in section 10.1

10 APPENDIX A: ARTIFACTS

10.1 URLs

- URL of GitHub Repository: https://github.com/awesomeaiden/595-regex-comp
- URL of Qualtrics Survey: https://purdue.ca1.qualtrics.com/jfe/form/SV_8AC0CQsRaNP3LTw
- URL of Reference Sheet: https://i.stack.imgur.com/BBPrT.png (Also shown in figure 13)

10.2 Survey Questions

- **String A:** Enter a string that fits this regular expression: `^[A-Z]{3,5}[0-9]*$`
- **String B:** Enter a string that fits this regular expression: `^((aab)*(c|de*))+$`
- **String C:** Enter a string that fits this regular expression: `^f(v*|(c|q))nt?$`
- **String D:** Enter a string that fits this regular expression: `^\\d+ \\+ \\d+[x-z] \\+ \\d+[x-z]\\^2 = [x-z]$`
- **Create A:** Create a regex that matches a string that starts with exactly 2 numbers, followed by an underscore, and ends with 4 capital letters. For example: `` `46_MKLN' ``.

Question	Control	RegExr	RegExper	Grex
String A	3	3	3	3
String B	3	3	3	3
String C	3	3	3	3
String D	3	3	3	3
Create A	3	3	3	3
Create B	3	3	3	3
Create C	3	3	3	3
Create D	3	3	3	3
Modify A	3	3	3	3
Modify B	3	3	3	3
Modify C	3	3	3	3
Modify D	3	3	3	3

Table 4: Number of times each question appeared in each survey section

- **Create B:** Create a regex that matches any non-empty string comprised of only numbers and underscores. Accepted example strings include `'124_1_'`, `` `__124' ``, `` `_' ``, and `` `123' ``.
- **Create C:** Create a regex that matches any string which starts with the word "userid", followed by an underscore (`'_'`), and then a sequence of 10 numbers. For example: `'userid_1264829527'`.
- **Create D:** Create a regex to match a date and time in the format: yyyy/mm/dd. For example: `` `2022/03/01' `` or `` `1992/10/07' ``. You may assume that every month can have up to 31 days.
- **Modify A:** Update the following regular expression to accept both '.com' and '.edu' email addresses:

  ```
  ^[A-Za-z0-9_]+@[\\w]+\\.com$
  ```

- **Modify B:** Update the following regular expression to accept phone numbers with or without a U.S. country code:

  ```
  ^(\\d\\d\\d) \\d\\d\\d-\\d\\d\\d\\d$
  ```

  For example: '+1 (123) 456-7890'.
- **Modify C:** Update the following regular expression to optionally accept a 4-digit postal code extension:

  ```
  ^\\d\\d\\d\\d\\d$
  ```

  For example: '47906-3518'.
- **Modify D:** Update the following regular expression to only accept first and last names up to and including 50 characters in length:

  ```
  ^[A-Za-z]+ [A-Za-z]+$
  ```

10.3 Question Distribution

11 APPENDIX C: POSTMORTEM

Over the course of this project, we have ran into our fair share of challenges, and difficulties, but also successes. This appendix was written to document those lessons and provide some insight into what might be the cause, as well as potential solutions.

11.1 Reflecting on scope

Our initial project layout and scope, as described in our Project Proposal document submitted at the beginning of the semester, differs in some ways from the final resulting experiment that we completed. It is worth mentioning that at the very start of our project we lost a member of our group due to them dropping the course. This was around the time we submitted our Project Proposal, and though it didn't very much change the scope of our proposal at the time, it did cause stress amongst the team, especially since we did not learn that they had dropped the course until several weeks of no-responses from that team member. The Microsoft Equivalent Code Tool is another thing that didn't go quite as well as we had originally hoped (as will be later discussed) and thus we had to remove its integration from the scope of our study. Moving on from the tool was difficult as it offered interesting functionality, but it was an milestone for us as we were able to move on to a more easily integrated tool like Grex. Overall, while the loss of the Microsoft Equivalent Code Tool was somewhat disappointing, we were still able to obtain very interesting results that addressed our research questions.

11.2 What went well

Throughout our project we learned many things that worked positively for the team, as well as things that could have been improved. To begin, some of the logistical choices in terms of communication and file storage technologies worked very well for us. We have been using a Discord channel created for our group, which has allowed us to have dedicated message boards for team-wide communication, and voice calls with screen-sharing as necessary for our meetings. In addition, we decided early on that we would store our files using Purdue's Office365 and a shared folder in OneDrive amongst our team members, which has worked great as a common point for the many versions of papers, presentations, and notes we have accumulated over the course of the semester. Finally, GitHub is the most obvious (and effective) choice for software version control, and was ideal for our survey development. Another thing that I believe went well was our execution of the survey with our study participants. We were able to successfully collect quantitative and qualitative data, as well as screen recordings for all 12 of our participants. Our food and gift card incentives were effective in attracting eager software engineering students to our study.

11.3 What didn't go well

As in every project, not everything went smoothly for us. One of the ways our team ran into difficulties was in our development of our survey webpage. Not everyone on our team had experience with web development (frontend or backend). Since a significant portion of the work for this project came in the form of full-stack software development, it became difficult for inexperienced members of the team to contribute and placed more of a burden on those who did have expertise in the area. Everyone attempted to contribute as they could, whether that be towards the technical progress or rather some administrative work, but since our timeline for several weeks in the middle of the semester consisted of only development work, the levels of contributions between members became skewed. Another thing that caused occasional issues for us

were some communication challenges. On large assignments such as the Project Proposal, revised Project Proposal, and their associated in-class presentations, we found it to be crucial that our team finds time to work simultaneously to ensure we were collaborating effectively and creating a cohesive result. Several times throughout the semester, especially in the days before these large assignments, our individual availability was not aligned. This resulted in a few late night calls with the team, especially on weekends, and generally left a feeling of exhaustion regarding our project for several days following.

11.4 Failures

There were a few significant difficulties and failures that our team ran into over the course of our study. First of all, one of the initial motivations behind our project was a tool provided by Microsoft, which we referred to as the "Microsoft Equivalent Code Tool." This tool, provided in the .NET Preview 7.1, takes in a regular expression and outputs a block of code, complete with comments in English, that implements the expression that was taken as input. Our goal was to incorporate this tool as one of our 3 tool options, in fact it was the 1st of the 3 tools we had agreed to incorporate into our project, going back as far as the initial project description and sketch. Initially we had scoped that we could host this tool in our backend server and pipe the output to the frontend during the survey as the participant interacts with it. However, the Equivalent Code tool does not operate as a traditional application in and of itself, but rather takes in a regular expression as input as part of a C# file, which is then compiled, and the equivalent code is output from the compilation process as part of a new, temporarily generated C# file. To use this tool in our survey, it would require us to take input from the participant and place the text inside a function call in a C# file, compile it, and then pipe only the relevant output back to the user. The complexity of the .NET framework and toolchain, and the difficulty of integration with our survey did not give us much slack in our development time. As we fell behind our proposed schedule, we made the decision to cut out the Equivalent Code tool and move forward with Grex, losing the unique approach the equivalent code tool would have provided. Still, substituting in the Grex tool was necessary for the success of the project, and still was a unique tool that provided different strategy and information about what makes a regular expression comprehension tool effective.

Another failure that we experienced during our project has to be the Internal Review Board (IRB) Approval Process. From the beginning, we proposed an experiment with human participants as the main part of our research study, and thus from early on we knew we most likely had to go through the process of having our work approved by the IRB. Though some of us were able to quickly complete the IRB training through the CITI course online, we ran into some difficulties organizing ourselves around the IRB process and of our own oversight lagged behind in the submission of our IRB application. Our initial submissions were returned to us with comments from the IRB advisors at Purdue. Though we believed we had addressed the requests placed in the submissions, we were repeatedly denied, and both the reviewers and our frustrations seemed to grow as there was a clear disconnect of understanding between us and the IRB regarding the scope and facilitation of our experiment. After weeks of back and forth in the Cayuse IRB system, with no progress seemingly being made, a member of our team reached out the to IRB directly, setting up an in-person meeting with several advisors. Through the completion of this 30-minute meeting we were able to easily reach an understanding and clear up the confusion regarding the treatment of our human participants. Within a few days time from this meeting we received IRB exemption and were clear to facilitate our human-subjects experiment. From this, we learned that face-to-face live communication is the best option in situations like this, and we could have saved weeks of IRB review time if we had simply scheduled this meeting sooner.

11.5 Lessons for the future

Starting with the Microsoft Equivalent Code tool, the lesson we can learn from this is to address the complexity as early as possible. It was known relatively early on, especially with our team's initial idea of hosting our survey on an Ubuntu machine, that integrating this tool was going to be a point of complexity for our project. However, our team waited until the development portion of our schedule to begin investigating. Several weeks into development, as we learned more about how the tool operates, we realized it might not be feasible for our project and timeline, requiring us to pivot to ensure a successful study. In future projects, addressing the points of difficulty earlier rather than later would leave us with either more ability to maneuver or a reallocation of effort to ensure the complexity is addressed.

With the Internal Review Board difficulties our team dealt with, the lessons we can learn from this are 2-fold. First of all, and similarly to the Microsoft Equivalent code tool, the IRB review work should have been addressed and prioritized earlier on in the semester. Since the length and timeline of the review is not directly in our control, it would have been advantageous to get a head start, especially since it's noted that the IRB submission are reviewed in the order they are received and can generally take multiple weeks. Had the IRB been extremely busy during our submission period it is possible that we wouldn't have been left with enough time to organize our experiment. The 2nd lesson we can take from the IRB process is that sometimes communication channels are limiting. There was a fundamental misunderstanding of the scope of our experiment during the initial IRB form submissions due to the IRB member's unfamiliarity with Regular Expressions and the data that we would be looking to collect. After meeting with them face-to-face, the matter was resolved rather quickly and we were able to gain approval to run our experiment.

One final lesson that could be taken from this experience is that sometimes the simplest solutions are the best solutions. Ultimately, the functionality of our study could have been accomplished with an off-the-shelf survey software. While we originally wanted the experience to be self-contained and self-tracking, external resources and screen recordings were eventually required to conduct the study, even after 6 weeks of development on the experimental tool. Additionally, because the software was almost fully custom, it was immature, and unexpected bugs were an issue. Perhaps in a future study, the tool could be matured further and it would have more benefits. In this study however, if we had chosen a simpler solution from the start we still could have created a streamlined

experience while allocating more time towards experimental design, data collection, and data analysis.

In our team's future endeavors in academia and industry we hope to build off of both the successes and failures experienced during this study.