

UC Merced

Proceedings of the Annual Meeting of the Cognitive Science Society

Title

Does Surprisal Predict Code Comprehension Difficulty?

Permalink

<https://escholarship.org/uc/item/50b308j9>

Journal

Proceedings of the Annual Meeting of the Cognitive Science Society, 42(0)

Authors

Casalnuovo, Casey

Devanbu, Prem

Morgan, Emily

Publication Date

2020

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed

Does Surprisal Predict Code Comprehension Difficulty?

Casey Casalnuovo (ccasal@ucdavis.edu), Prem Devanbu (ptdevanbu@ucdavis.edu)

Department of Computer Science, University of California, Davis
One Shields Avenue, Davis, CA 95616 USA

Emily Morgan (eimorgan@ucdavis.edu)

Department of Linguistics, University of California, Davis
One Shields Avenue, Davis, CA 95616 USA

Abstract

Recognition of the similarities between programming and natural languages has led to a boom in the adoption of language modeling techniques in tools that assist developers. However, language model *surprisal*, which guides the training and evaluation in many of these methods, has not been validated as a measure of cognitive difficulty for programming language comprehension as it has for natural language. We perform a controlled experiment to evaluate human comprehension on fragments of source code that are *meaning-equivalent* but with *different surprisal*. We find that more surprising versions of code take humans longer to finish answering correctly. We also provide practical guidelines to design future studies for code comprehension and surprisal.

Keywords: Code Comprehension; Language Model Surprisal; Transformer Model

Introduction

As software has become nearly ubiquitous, people, even those who do not consider themselves to be developers, interact with code and learn to program. One of the largest costs in software engineering comes from maintaining existing code (Banker et al., 1993); understanding code that others have written (or returning to code a person has written themselves) takes up a large portion of a programmer’s time (Tiarks, 2011).

Though code comprehension research has a long history—see Johnson & Soloway (1985); Mayer (2013) for early examples and Siegmund (2016) for a survey of the area—there have been more recent calls by psycholinguists to explicitly understand the cognitive processes that drive programming (Fedorenko et al., 2019). Code is often treated as just another kind of language; people refer to teaching coding in terms of “literacy” and describe its structure using terms from linguistics such as grammar, syntax, and semantics. This acknowledgment of the human communicative element of programming is not new: Knuth’s Literate Programming emphasizes writing code not for machines, but for the benefit of other developers (Knuth, 1984). While the degree to which natural and programming language share cognitive processes is unknown, recent work has shown some regions of the brain used in natural language processing are also used in programming (Siegmund et al., 2014). In contrast, eye-tracking studies have shown people read code differently than natural language; their eyes jump around non-linearly, and this effect increases with experience (Busjahn et al., 2015).

Appreciation of the similarities and differences between natural and programming languages has led to the adoption of computational language models for code. Language models operate by learning probability distributions over text, and key to training these models is the notion of *surprisal*. The surprisal of a token is its negative log probability in context as captured by the language model¹. Programming language is more repetitive and has lower surprisal than natural language, i.e. language models find code *more predictable than natural language* (Hindle et al., 2012). Moreover, this finding holds across many natural and programming languages, and some of this is contingent on human choices, above and beyond that which would be expected by inherent differences in language structure (Casalnuovo, Sagae, & Devanbu, 2019). This potentially makes language models *even more effective in a source code context* and has led to their adoption for many applications. These applications include automatic completion of code, finding defects and errors, or generating documentation from code, all of which are tools making a real impact—see Allamanis et al. (2018) for an extensive survey. These tools often leverage language model surprisal during training; by minimizing surprisal they are assumed to be better models of source code.

Though these tools and methods have obtained wide acceptance, the underlying measure of surprisal used by language models has thus far seen very little validation as relating to what makes code “better” for humans. This lack of testing of assumptions and validating tools is a longstanding problem for code comprehension (Siegmund, 2016). For example, often times when assessing code readability, experiments have relied on developers’ *perception* of code, which may not be representative of how easy the code is to actually understand (Scalabrino et al., 2017). In natural language, the relationship between language model surprisal and cognitive load is fairly established; higher surprisal leads to higher load (Levy, 2008; Smith & Levy, 2013), but this is not true for code. Surprisal’s impact in natural language and the evidence that human choices influence surprisal in code (Casalnuovo, Lee, et al., 2019; Casalnuovo, Sagae, & Devanbu, 2019) are suggestive of surprisal having an impact on human comprehension of code, but this still must be demonstrated.

¹This is closely related to *entropy*, which is the expectation of surprisal.

In code, this relationship is complicated by developers needing to simultaneously write code for two ‘audiences’—one audience is the machine, which obtains precise semantics through execution, and the other audience is other humans. These effects intermingle in complex ways, with some elements of code, such as variable names, whitespace formatting, or parentheses use typically possessing no meaning for the formal algorithmic channel that the machine processes; these elements communicate only to other humans (Casalnuovo et al., 2020).

Thus, for code, two expressions can have *identical computational meaning* (i.e. the meaning obtained by the machine when executing the code) but be written in different ways. For example, consider the statement $a = b/c * d$; This could be equivalently written as $a = (b/c) * d$; which clarifies the order of operations to the developer, but has no effect on the meaning to the machine. Similarly, consider how the common code idiom for incrementing is usually written as $i = i + 1$; and almost never as the entirely equivalent $i = 1 + i$; Developers may choose to write one over the other due to either readability concerns or possibly the pressures of existing style or convention.

This feature enables opportunities to explore surprisal in code via controlled experiments. By looking at source code snippets with different surprisal, but equivalent meaning, we can test hypotheses about whether surprisal can measure the readability and understandability of source code. Most related to this study is our previous work (Casalnuovo, Lee, et al., 2019), which looks at the relationship of *human preference* and surprisal. In that study, humans tended to prefer code with lower surprisal in a forced-choice experiment between two lines with different surprisal but identical computational meaning. We use a similar methodology and transformations in this study.

In contrast to the preference study, here we consider how surprisal influences *human comprehension* of source code. One way to measure if someone understands code is if they can execute it: given some input can they correctly describe the output? If we have two snippets of source code with equivalent meaning but different surprisal, we can ask humans to compute the outcome of each of them. We ask two primary research questions to see how easily they understood each variant: how *accurately* do humans compute the answer, and how *quickly* do they *correctly compute* the answer?

Methodology

Materials

Data To develop meaning-equivalent source code fragments, we first train a language model to predict the surprisal of code. Our training and test data come from the 1000 most starred Java open source projects on Github (<https://github.com/>). From this set, we selected a smaller sample of the 30 Java projects with the most opportunities to perform meaning preserving transformations. We split these into 21 and 9 projects for the training and testing set, chosen ran-

domly, with some preprocessing to remove potentially duplicate or highly similar files by removing those with identical filename and parent directories.

Language Model Training We use a Transformer Model (Vaswani et al., 2017) with Byte-Pair Encoding (BPE) (Sennrich et al., 2016) to train the model and obtain surprisal scores for each line. BPE creates subtokens of roughly similar frequency from the tokens, reducing vocabulary size to 9165 subtokens. Code has a larger vocabulary than natural language, which makes training with neural models difficult, and BPE has proven effective at addressing this vocabulary size problem (Karampatsis et al., 2020). This model is implemented in TensorFlow (Abadi et al., 2016), using 2 layers and dimensions of 512 for the attentional and feedforward layers, 8 attention heads, and a dropout rate of 0.1. We train for 10 epochs, with learning rate 0.2, a batch size of 15000, and 200 tokens per sequence.

Table 1: Pseudocode examples of transformations. The first column shows the general type of operation, second lists the operators involved, and the last two show an example.

Swap	*	$a * b$	$b * a$
Arithmetic	+	$a + b$	$b + a$
Swap	==, !=	$a != b$	$b != a$
Relational	<, <=, >, >=	$a <= b$	$b >= a$
Adding Parentheses		$a + b * c$	$a + (b * c)$
Removing Parentheses		$a + (b * c)$	$a + b * c$

Meaning-Preserving Transformations We use 4 categories of meaning-preserving code transformations. Table 1 shows all our transformation with examples in pseudocode. At the top, we have two types of operations swapping around arithmetic and relational operators. For the arithmetic swaps, we look at + and * operations, which are commutative, and conservatively swap only in cases using numerical variables and literals, avoiding expressions with functions as they may contain side effects that change code meaning. For relational operators, we swap the operands around ==, !=, >, >=, <, and <=. If the relation is not symmetric (less and greater than variants), we also invert the operator when swapping to maintain precise meaning. We also add and remove parentheses that are not essential to the meaning of the code, but are often added by developers for readability. These examples appear in the bottom two rows of Table 1.

Experimental Materials Selection and Validation We selected a total of 64 pairs of original and transformed lines of code (16 from each transformation) to present to study participants. We used a combination of initial random sampling and then manual selection, creating a set of pairs where half the time the transformation decreased surprisal relative to the original line, and half the time it increased surprisal. In the first phase, we randomly sampled 64 examples from the test set for each transformation. These samples came from be-

yond the median in both the positive and negative direction to ensure pairs had large surprisal differences. This initial sample was filtered to automatically exclude expressions that were likely to be automatically generated (e.g. hashes), contained rare operations (like bit shifts), were overly easy (comparisons to null or 1), or were over 80 characters long. From this initial set we manually selected 16 samples per transformation, so that no 2 examples would be too similar, and avoided samples that required too much contextual information (such as calls to anything but the most basic functions, like `size()` on a list).

Next, we artificially created concrete values to initialize all variables used in the expression. These initializations were used for both versions of the expression, and if there was more than 1 variable to initialize, the order was randomized. We used simple initializations for the variables in each expression, to reduce cognitive overhead, but also such that participants would have to evaluate the entire expression. Once these initializations were generated, we ran the code for each of the 64 pairs to verify the correct answer and that they were equivalent. Figure 1 shows an example of initialization and a pair of snippets.

```
int width = 7;
int x = 3;
// Original Source Code
if(x >= width / 3 * 2) {
// Transformed Source code
if(x >= 2 * width / 3) {
```

Figure 1: Example initialization and expression pair. Participants saw only one of the pair (without the Original/Transformed label) and the question: "Does the expression evaluate to true or false?"

Procedure

Our experiment consists of presenting each subject with 32 randomly chosen samples from our 64 pairs, sampling across transformations from all types. Subjects are randomly shown only 1 variant of the pair, to prevent any learning effect from seeing both versions. For each sample, the subject is first shown the variable initializations for 3 seconds, after which they can advance the page to see the expression that uses them in addition to the initializations.

Then, they are asked to mentally compute the value of the expression after execution. Tasks included computing a numerical value, determining whether a boolean expression was true or false, or determining how many times a `for` loop would execute. For numerical questions, respondents entered the value in a text box, and for boolean values they choose between true/false buttons. For text questions the cursor began in the text box, and true/false questions could be answered with the 't' and 'f' keys, enabling subjects to complete the experiment entirely using their keyboard.

During the experiment, we measured both answer correctness and response time. Correctness is straightforward,

though we give credit for similar answers (i.e. "8" counts for a question with a floating-point answer of "8.0"). For timing, we used the high precision timing Javascript library `Performance`² to record the timing of every keystroke and click involving the text box or answer buttons. Using these times we constructed two response variables of interest: *First Action Time* and *Last Action Time*. Respectively, these are the first and last times the subject interacts to answer the question (whether by click or keystroke), excluding the final click/keystroke to submit their answer. We tried two different timing measures as we were uncertain which would be most appropriate for this type of code comprehension experiment due to lack of prior literature. We hope the results here will help guide such experiments in the future.

We presented the survey to workers on Amazon's Mechanical Turk³. To qualify for the main experiment, subjects had to pass a 3 question Java code comprehension task. Answering all 3 questions correctly allowed them to choose to continue to the main task. After the subjects completed the main task, there was an optional demographics survey and a couple of open-ended feedback questions. At the end, they were presented their overall score on the main task, and we exclude from data analysis any response that received a score of less than 20 out of 32.

Participants

We restricted our participants to workers on Mechanical Turk who had 1000 or more hits, a 99% or greater acceptance rate, and were from the US or Canada. We also used Unique Turker⁴ to avoid duplicate attempts. In total, we had 343 attempts on the qualification task, and 116 full completions of the main task with 111 scoring 20 or higher.

The subjects reported age with mean 32.2 and s.d. 8.8 years, Java experience with mean 10.5 and s.d. 5.3 years, and took 34.2 minutes on average with s.d. 14.3. About 67% programmed at least a few times a week and most of the rest at least a few times a month. Almost all participants had at least some college education, with over 50% having a Baccalaureate degree. Most use Mechanical Turk as an extra source of income. We paid \$1 to everyone who took the qualification (pass or fail), and an additional \$4 to everyone who completed the main task, regardless of their score.

Results

Statistical Analysis

We have 3 primary response variables of interest, 1) a binary variable for whether the respondent answered the question correctly, 2) their *First Action Time* in seconds, and 3) their *Last Action Time* in seconds. Our primary explanatory variable for surprisal is a binary value which is 0 if this variant was the less surprising version or 1 if it is the more surprising

²<https://developer.mozilla.org/en-US/docs/Web/API/Performance>

³<https://www.mturk.com/>

⁴<https://uniqueturker.myleott.com/>

version. We analyze our data using mixed-effects regression models. Our full models contain fixed effects for which of the 4 transformation types a question was, and whether the question was a text box or true/false question. We also considered interaction effects for each of these with the surprisal value. Our random effects are the maximal structure justified by the design (Barr et al., 2013); for items, we have a random intercept and a slope for surprisal; for subjects, a random intercept and slopes for surprisal, transformation type, question type, and their interactions. We use deviation coding for all categorical variables; each coefficient is in comparison to the grand mean. Therefore, for example, the regression formula for our full model using the binary measure for *Last Action Time* is as follows: $Last\ Action\ Time \sim Surprisal*(TransType + QuestionType) + (1 + Surprisal*(TransType + QuestionType)|ResponseID) + (1 + Surprisal|Question)$. The correctness models were logistic regressions, and the timing models were lognormal, as we observed that fit the distribution of the response well. We fit these models using the *brms* package for bayesian regression, using default priors (Bürkner, 2017). We also tested each of these full models against simpler ones to check robustness, using WAIC scores to compare them (Watanabe, 2010). When the simpler models have qualitatively similar results to the full models, but much better WAIC scores, we present the simpler models.

As these experimental methods have not typically been applied to code, we additionally want to explore best practices for these types of experiments, so we also considered our data in a few other ways. We considered our models with and without timing outliers (cases where subjects answered more than 3 standard deviations away from the mean of first and last action time), but observed that they had minimal impact, so we present the models with outliers. We also modeled the text answer and true/false questions separately, after observing different behavior from these questions in our models and plots. Supplementary materials, data, and R notebooks showing models and plots not included in results are available at: <https://doi.org/10.5281/zenodo.3626129>

Timing and Surprisal

Table 2: Fixed effects for bayesian mixed-effects lognormal regression comparing if a variant was more or less surprising against the *Last Action Time*. WAIC scores suggested the model without interactions was best.

	Estimate	Error	1-95% CI	u-95% CI
Intercept	3.08	0.06	2.96	3.20
Surprisal	0.07	0.02	0.02	0.12
AddParen	0.09	0.09	-0.08	0.26
Arithmetic	-0.20	0.09	-0.37	-0.03
Relational	0.02	0.09	-0.17	0.19
Text	0.15	0.05	0.04	0.26

First, let's consider the question of whether more surprising code takes longer for humans to comprehend. We will focus

primarily on *Last Action Time*, as the effects were larger and more significant than *First Action Time*. Figure 2a shows the median time difference for *Last Action Time* plotted against the difference in surprisal. Note that though the models and analysis presented here and for the correctness results operate on the full data with random effects, the figures show summarized versions of the data with simple regression lines to aid in visualizing the data. These simplified models trend upwards as predicted; participants tended to answer questions about the more surprising code variant more slowly.

Now, to see if these effects are significant, we present the results of our mixed-effects models which account for the variance of the questions and subjects. We will discuss the *Last Action Time* model in detail in Table 2, and then briefly mention the *First Action Time* model. Surprisal has a significant effect, and as the regression is lognormal, we can interpret the average variant with lower surprisal taking 21.8 seconds and with higher surprisal taking 23.3 seconds. Looking at the other coefficients, we see significant differences on arithmetic questions and on text questions, with arithmetic being significantly faster and text questions significantly slower. In comparison to this model, the estimated surprisal coefficient for the *First Action Time* model with the best WAIC score is 0.06 with 95%-CI (0.00, 0.11). The effect is suggestive, but not large enough to conclude significance.

When we divide our data to look at the text and true/false questions separately in models, we see that the effect is less significant for the true/false models, but much more so in the text models. The text questions exhibit larger effects; in fact, for the text questions only both first and last action time show significant effects of surprisal: 0.12 with CI (0.01, 0.23) for the first action model and 0.14 with CI (0.07, 0.21) for the last action model. However, when modeling the true/false questions only, the effects are no longer significant - both contain 0 within their credible intervals. Figure 3a shows these trends with summarized data using simple regression models.

Case Studies Some transformations lead to drastic changes in the time to answer correctly. Using the difference in median *Last Action Time*, we present the most extreme examples for and against our hypothesis that higher surprisal leads to longer mental computation. For the most extreme example that agreed with our theory, we saw an increase of 22.5 seconds when changing `time -= hours * 60 * 60;` to `time -= 60 * hours * 60;`. The language model preferred the original in this case. The most extreme change in the direction opposite to what is predicted by our theory also happened when the language model gave lower surprisal on the original, but the transformation proved much easier to comprehend. After adding parentheses around `step >= minStep` in the line `for(int i = 0; i < _maxfev && step >= minStep; ++i, step *= _stepDec)`, subjects were able to correctly determine how many times the loop executed a median of 26.1 seconds faster. We theorize that the unusual nature of this loop construction might explain why surprisal and

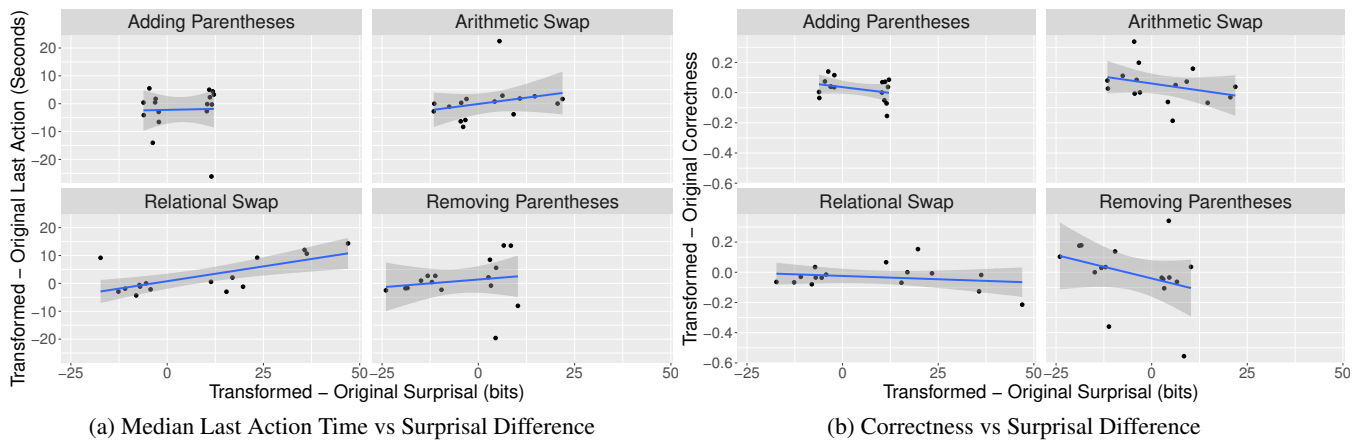


Figure 2: Per question pair plots of a) last action time and b) question accuracy differences against the difference in surprisal between the transformed and original code broken down by each of the 4 transformations. For the x-axis on each plot, further left indicates when the original code is more surprising, and further right means the transformed code is more surprising. For a), the y-axis plots the difference in median time between the two; smaller values (below 0) indicate that questions about the transformed code were answered more quickly. For b) the y-axis plots the difference in the fraction of correct answers. Higher values mean more people correctly answered questions about the transformed code.

comprehension were unaligned in this case. Regardless, these examples show that small line-level transformations can *substantially* change comprehension time.

Accuracy and Surprisal

For our mixed-effects model of correctness, the best model judging by WAIC scores was a model with no interactions that excluded the transformation type as a categorical variable. The binary surprisal coefficient was -0.21, suggesting a negative trend between surprisal and correctness. However, the 95% credible interval on this coefficient was (-0.56, 0.15). As this interval is wide and includes 0, we cannot conclusively say that the effect is significant, though it is at least in the expected direction. Graphically, we express this trend in Figure 2b, comparing the accuracy against the surprisal difference, summarized for each question.

As in the timing models, there is a significant difference between the text and true/false questions, so we divide the data and model them separately. We see that the effect is even less significant for the true/false models, but more so in the text models. The coefficient for surprisal for the text-only model is -0.37, but the 95% credible interval still contains 0 (-0.87, 0.11). Figure 3b shows a summarized plot of these trends. Therefore, we can at best say the effects are suggestive but not conclusive for correctness; further study is needed to link surprisal and comprehension accuracy.

Case Studies As with timing, we present the most extreme cases for and against our hypothesis that higher surprisal means fewer correct responses for the questions. For an example of the largest change in the expected direction, we swapped the operands in the expression `res[numstart + i] += scale * numVals[i];` to change

it to `res[numstart + i] += numVals[i] * scale;`. The transformed code had lower surprisal, and it improved the percent of correct answers from 55.5% to 89.2%. In this case, we theorize that grouping the array accesses together might have made the mental computation easier. The most extreme change in the unexpected direction was the transformation from `return (2.0 / sampleSize) * (prediction - lb);` to `return 2.0 / sampleSize * (prediction - lb);`. The transformation had lower surprisal, but 93.1% of our subjects correctly computed the original code and only 57.1% did so for the transformed version. Perhaps the lack of parentheses made the order of operations between the divide and multiply operations unclear. Therefore, as with comprehension time, we can find examples where small transformations have large impacts on correct comprehension.

Discussion

Does higher language model surprisal predict increased difficulty in processing code? Our experiment provides suggestive but not definitive evidence for this effect. The clearest effects appeared in the models measuring the total time to answer the comprehension questions; for models measuring the first time the subjects interacted with the questions and whether the question was answered correctly, the effects trended in the predicted direction but were not significant. Exploratory analysis beyond our main models showed these effects were more pronounced when we only considered text box questions and excluded true/false questions.

One may question whether such small changes to single lines of code could really affect how easy they are for humans to understand. However, our case studies demonstrated this

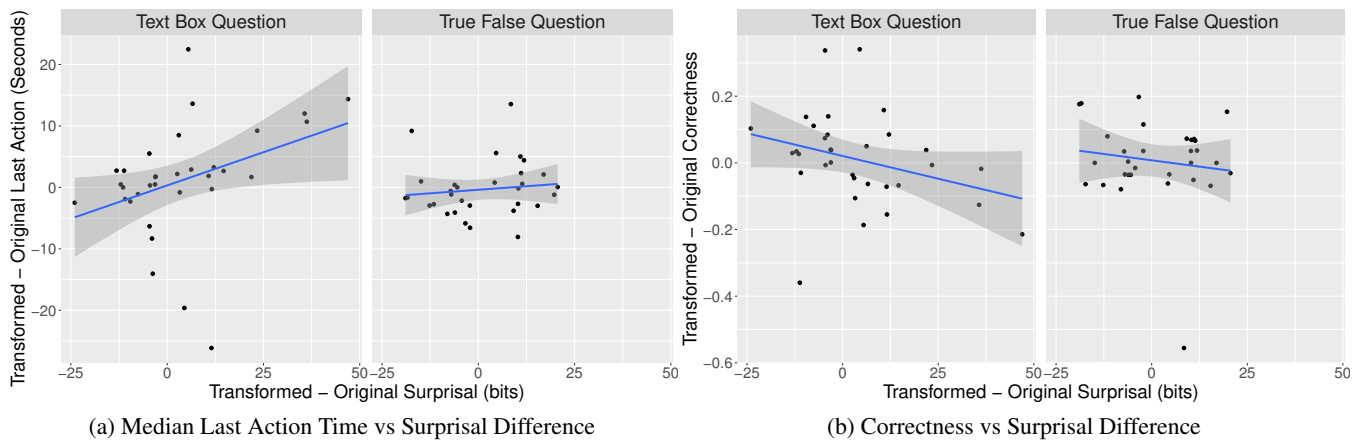


Figure 3: Per question pair plots of a) last action time and b) question accuracy differences against the difference in surprisal between the transformed and original code broken down by each of 2 types of questions. For the x-axis on each plot, further left indicates when the original code is more surprising, and further right means the transformed code is more surprising. For a), the y-axis plots the difference in median time between the two; smaller values (below 0) indicate that questions about the transformed code were answered more quickly. For b) the y-axis plots the difference in the fraction of correct answers. Higher values mean more people correctly answered questions about the transformed code.

is not true by counterexample; we found cases where even a small change could drastically alter how quickly and accurately participants could answer questions. A single changed parenthesis or reordered statement can lead to a misunderstanding about some code’s meaning.

There may be additional factors explaining when surprisal does or does not predict comprehension well. If language models learn from past exposure, the alignment between surprisal and comprehension may depend on the type and frequency of similar expressions. For instance, in chess, human experts were able to more quickly comprehend and remember known configurations, but displayed ability little better than novices when exposed to unusual random configurations (Chase & Simon, 1973). Sorting out the relationship of surprisal with frequency and other potential abstract factors could also be a promising future area of research, as both have been shown to play a role in natural language preference (Morgan & Levy, 2016). In more novel code expressions, does surprisal relate to comprehension, and if so, what factors might explain this connection?

Our experiment shows comprehension tasks are a viable method of studying how people process code and provides some recommendations for future studies. First, focus on comprehension questions that require text entry answers. True/false questions can be more easily guessed and might be too simple to see the desired effects. Next, more difficult questions or situations may show more significant effects. For example, surprisal has also been used as a means to drive code *obfuscation* (i.e. transforming code to make it harder to interpret) (Liu et al., 2017). Though the authors of this paper focused more on inhibiting machine learners than human comprehension, establishing if surprisal driven obfuscation is

similarly effective on humans could help create better obfuscation tools for privacy and intellectual property protection. In other contexts, we hope that as these effects are better understood, it may be possible to use surprisal as a method to guide automated tools modifying code to be more easily understandable by humans without altering its computational meaning. Such tools could include style recommendations or parts of the code editor. Finally, methods of measuring readability and comprehension of code may support more effective teaching of coding, establishing good code writing practices that are more understandable to existing programmers.

Acknowledgments

We would like to acknowledge the assistance of students and postdocs who helped in testing and providing feedback on our comprehension task, including Vincent Hellendoorn, Kevin Jesse, Anand Sawant, Toufique Ahmed, and Skyler Reese. We would also like to acknowledge the funding support of NSF grant #1414172: SHF: Large: Collaborative Research: Exploiting the Naturalness of Software.

References

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... others (2016). TensorFlow: A System for Large-Scale Machine Learning. In *OSDI* (Vol. 16, pp. 265–283).
- Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys (CSUR)*, 51(4), 1–37.
- Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, 36(11), 81–95.

- Barr, D. J., Levy, R., Scheepers, C., & Tily, H. J. (2013). Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, 68(3), 255 - 278.
- Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on* (pp. 255–265).
- Bürkner, P.-C. (2017). brms: An R Package for Bayesian Multilevel Models Using Stan. *Journal of Statistical Software*, 80(1), 1–28.
- Casalnuovo, C., Barr, E. T., Dash, S. K., Devanbu, P., & Morgan, E. (2020). A Theory of Dual Channel Constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.
- Casalnuovo, C., Lee, K., Wang, H., Devanbu, P., & Morgan, E. (2019). Do People Prefer “Natural” code? *ArXiv, abs/1910.03704*.
- Casalnuovo, C., Sagae, K., & Devanbu, P. (2019). Studying the Difference Between Natural and Programming Language Corpora. *Empirical Software Engineering*, 24(4), 1823–1868.
- Chase, W. G., & Simon, H. A. (1973). Perception in Chess. *Cognitive Psychology*, 4(1), 55–81.
- Fedorenko, E., Ivanova, A., Dhamala, R., & Bers, M. U. (2019). The Language of Programming: A Cognitive Perspective. *Trends in Cognitive Sciences*, 23(7), 525–528.
- Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2012). On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 837–847). Piscataway, NJ, USA: IEEE Press.
- Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-Based Program Understanding. *IEEE Transactions on Software Engineering*(3), 267–275.
- Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., & Janes, A. (2020). Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. In *42nd International Conference on Software Engineering (ICSE '20)*. ACM. doi: <https://doi.org/10.1145/3377811.3380342>
- Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2), 97–111.
- Levy, R. (2008). Expectation-based syntactic comprehension. *Cognition*, 106(3), 1126 - 1177.
- Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M., & Sun, J. (2017). Stochastic Optimization of Program Obfuscation. In *Proceedings of the 39th International Conference on Software Engineering* (pp. 221–231).
- Mayer, R. E. (2013). *Teaching and Learning Computer Programming: Multiple Research Perspectives*. Routledge.
- Morgan, E., & Levy, R. (2016). Abstract knowledge versus direct experience in processing of binomial expressions. *Cognition*, 157, 384–402.
- Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., & Oliveto, R. (2017). Automatically Assessing Code Understandability: How Far Are We? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (pp. 417–427).
- Sennrich, R., Haddow, B., & Birch, A. (2016, August). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1715–1725). Berlin, Germany: Association for Computational Linguistics.
- Siegmund, J. (2016). Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 5, pp. 13–20).
- Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... Brechmann, A. (2014). Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 378–389).
- Smith, N. J., & Levy, R. (2013). The effect of word predictability on reading time is logarithmic. *Cognition*, 128(3), 302–319.
- Tiarks, R. (2011). What Maintenance Programmers Really Do: An Observational Study. In *Workshop on Software Reengineering* (pp. 36–37).
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems* (pp. 5998–6008).
- Watanabe, S. (2010). Asymptotic Equivalence of Bayes Cross Validation and Widely Applicable Information Criterion in Singular Learning Theory. *Journal of Machine Learning Research*, 11(Dec), 3571–3594.