COGNITIVE SCIENCE
A Multidisciplinary Journal

Cognitive Science 44 (2020) e12921

ISSN: 1551-6709 online
DOI: 10.1111/cogs.12921

Do Programmers Prefer Predictable Expressions in Code?

Casey Casalnuovo,[a] Kevin Lee,[a] Hulin Wang,[a] Prem Devanbu,[a]
Emily Morgan[b]

[a]*Department of Computer Science, University of California, Davis*
[b]*Department of Linguistics, University of California, Davis*

Received 21 March 2020; received in revised form 15 October 2020; accepted 19 October 2020

Abstract

Source code is a form of human communication, albeit one where the information shared between the programmers reading and writing the code is constrained by the requirement that the code executes correctly. Programming languages are more syntactically constrained than natural languages, but they are also very expressive, allowing a great many different ways to express even very simple computations. Still, code written by developers is highly predictable, and many programming tools have taken advantage of this phenomenon, relying on language model *surprisal* as a guiding mechanism. While surprisal has been validated as a measure of cognitive load in natural language, its relation to human cognitive processes in code is still poorly understood. In this paper, we explore the relationship between surprisal and programmer preference at a small granularity—do programmers prefer more predictable expressions in code? Using *meaning-preserving transformations*, we produce equivalent alternatives to developer-written code expressions and run a corpus study on Java and Python projects. In general, language models rate the code expressions developers *choose* to write as more predictable than these transformed alternatives. Then, we perform two human subject studies asking participants to choose between two equivalent snippets of Java code with different surprisal scores (one original and transformed). We find that programmers *do* prefer more predictable variants, and that stronger language models like the transformer align more often and more consistently with these preferences.

Keywords: Surprisal; Language models; Dual channel constraints; Source code expressions; Meaning-preserving transformations; Human preference

1. Introduction

When we describe programming, we often use terms like "reading" and "writing" to explain how a person interacts with code, as if it was another form of natural language.

Correspondence should be sent to Casey Casalnuovo, Department of Computer Science, 2063 Kemper Hall, One Shields Avenue, Davis, CA 95616. E-mail: ccasal@ucdavis.edu

Moreover, programming is becoming universal; diverse groups of people make use of code in their lives, and the skills it provides are recognized by society; for example, 9 out of 10 parents in the United States want their children to learn programming (Google & Gallup, 2016). As coding proficiency moves from a specialized to a general or even expected set of skills, comparisons to natural language have led to calls to frame coding skills in terms of literacy (Vee, 2017). However, despite these comparisons, it is not clear in what ways humans process programming languages similarly to or differently from natural languages.

Though code is often conceived as a form of communication with the computer, acknowledgment and emphasis on the importance of code as a *human-to-human* communication medium has long existed in the computer science community. For a classical example, Knuth's "Literate Programming" makes a call for just such an emphasis: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do" (Knuth, 1984). Maintaining code is the most costly component of application development, with reading and understanding code taking up the majority of a programmer's time (Banker, Datar, Kemerer, & Zweig, 1993; Tiarks, 2011). As such, *how* code is written matters beyond just *what* it computes. Certainly, team-developed software must be readable by all the team participants; but even when code is authored and maintained by a single programmer, they themselves are still an audience, needing to read, understand, and maintain the code.

If code is another form of human communication, albeit one subject to different constraints, how do these constraints affect how code is written and understood? How is cognitive effort expended in this environment, and how does that compare to how it is expended in natural language? In natural language, one common lens for relating cognitive effort to text is *predictability*; more predictable text conveys less information to the reader and is easier to process. For example, consider filling in the blank for "The journey of a thousand miles begins with a single ___" versus "You wouldn't believe that I saw a yesterday ___." As a well-known proverb, the first is highly likely to be completed with "step", whereas the second could be any number of options. Likewise, in code, the common idiom for iterating through values "`for(int i = 0; i < n; ____)`" would likely be filled with the common iterator incrementing expression "`i++`," whereas completing a conditional that operates on the value a function returns (which could be many) "`if(returnCode ==____)`" could be challenging.

It is possible to get a sense of predictability in each of these examples, but despite the similarities between code and natural language, (correct) code is a far more constrained form of communication. Code must be executed, and errors prevent a program from running correctly or even running at all; in contrast, humans can use "noisy channel" models to correct sentences to likely correct interpretations (Ferreira & Patson, 2007; Levy, 2008b; Levy, Bicknell, Slattery, & Rayner, 2009).

In natural language, it has long been established that more predictable words are easier and faster to process than less predictable words (e.g., Ehrlich & Rayner, 1981; Kutas & Hillyard, 1984). But the same has not been established in code. In the constrained

environment of code, to what degree do humans have a sense of code's predictability, and does it drive how code is written? Most essentially, do programmers prefer predictable code? We aim to establish a similar understanding of predictability in code as exists in natural language.

1.1. Predictability in code and in natural language

Surprisal is often used to measure and operationalize predictability, and it can be applied to natural language or code. Specifically, the surprisal of a word (or a token of code) is its negative log-probability in context, where the context may include the previous or surrounding tokens, or potentially be even more general. Surprisal uses a probability distribution to measure how likely words and sentences (or tokens and statements) are in a given language. This probability distribution can be created in a few ways, from being built by human estimates or (more scalably) estimated from a *statistical language model* learning it from a corpus.

In natural language, it has been established that reading times are proportional to the surprisal of a word in context across many orders of magnitude of probability (Demberg & Keller, 2008; Hale, 2001; Levy, 2008a; Smith & Levy, 2013), establishing a link between cognitive effort and predictability. How does predictability, measured via surprisal, operate in the more constrained environment of source code? Most superficially, are code statements with low surprisal preferred over and easier to process than very surprising ones? Addressed with greater nuance, it would also be useful to understand whether small degrees of surprisal matter in code, whether there are contexts where surprisal is *not* associated with human preferences and comprehension, and what factors determine these contexts.

As a secondary consideration, answering these questions is important from an applications perspective as surprisal has been used within practical tools that model code, but it has not thus far been validated as a cognitive measure of code preferences or comprehension. A key result by Hindle, Barr, Su, Gabel, and Devanbu (2012) found that language models trained on source code were *far* more predictable and repetitive than those trained on natural language. Using classical *n*-gram models of code, they found that the entropy of code written in Java is around 3 to 4 bits lower than that of English, suggesting that Java is *8 to 16 times more predictable than English* (Hindle et al., 2012). This demonstration of the predictability of code has inspired hundreds of papers applying such models to code for the purposes of code completion, defect finding and repair, automatic documentation and summarization, and many more (see the survey by Allamanis, Barr, Devanbu, & Sutton, 2018).

This finding raises questions about why Java (and potentially code in general) is more predictable than natural language. Does it relate to syntactic or grammatical constraints, or is it the result of humans choosing to write code more predictably? Casalnuovo, Sagae, and Devanbu (2019) studied this question across a wide variety of programming and natural languages, persistently finding code more predictable. In experiments removing syntactic closed category words (prepositions, determiners, conjunctions, pronouns, etc.) and

looking at parse ambiguity, they found these differences could not entirely account for why code was more predictable (in fact, the content words of code are relatively even *more* predictable than in English). Likewise, they examined a wide variety of specialized English corpora (technical, legal, recipes, foreign language learners) where a desire for higher precision or increased difficulty might cause humans to write more predictably. Compared to generalized—or even domain (a collection of books in the science fiction genre) or author (a collection of the works of Shakespeare) limited—English, these texts exhibited greater repetition and predictability, albeit not as much as code itself. Between the studies comparing code-like corpora and syntactic manipulations, the paper concluded that at least some of the predictability of code must be *contingent on human factors/choices*, and not simply arising from inherent grammatical constraints.

Therefore, our goal is to test if humans have a sense of predictability in code, and whether preference for predictable forms (driven by ease of production and/or ease of comprehension) drives some of the code's highly predictable nature. The previous study by Casalnuovo et al. (2019) has two major limitations. First, the study lacks a direct comparison of surprisal to human judgments of code. Do humans actually prefer less surprising code or does it not matter? Second, the study operates at the level of entire corpora. There are many different types of statements in code with many different meanings; an approach comparing code segments with *equivalent* meaning could further highlight the relationship between surprisal and human preferences. Here, the formal nature of code enables the creation of expressions that have exactly the same computational meaning, while having *different surface forms*. Next, we describe a framework that shows how these equivalent constructions can be formed as a result of how code meaning must be communicated to two audiences: human and machine.

1.2. Dual channel constraints and equivalent meaning in code

The theory of *dual channel constraints* (Casalnuovo, Barr, Dash, Devanbu, & Morgan, 2020) describes how programmers express meaning in code to each audience. In this formulation, code is comprised of two channels—an algorithmic (AL) channel and a natural language (NL) channel. The AL channel is the computed meaning of the code; it is the only part of the code communicated between the developer and the machine, which lacks the ability to utilize any of the NL signals of the code. The NL channel consists of the natural language elements of code latent in comments, variable names, and coding styles. The machine audience is only aware of only the AL channel, but humans are aware of *both channels*, creating an asymmetrical relationship in the information passed to the two audiences. Thus, the theory states that programmers are aware of, and coding for, each of these two audiences *simultaneously* when writing, which imposes constraints on each channel. Programmers must both correctly and precisely implement the computable meaning while also expressing that meaning in a way that is easy to read and write. These constraints and the asymmetrical nature of the channels is a useful framing from an applications perspective (e.g., using them to improve typing systems in programs; Dash, Allamanis, & Barr, 2018).

However, this constrained environment also offers opportunities for experiments that are difficult in natural language. In particular, the formal semantics of the AL channel provides the affordance to manipulate NL forms, while preserving meaning, in a way that purely natural language does not. Programmers themselves have exploited this property in creative ways. For example, people have used code for artistic expression *via* esoteric or "weird" languages, such as the language Shakespeare that makes all programs written in it resemble parodies of stage plays, or by manipulating existing popular languages, such as in the International Obfuscated C Code Contest (Broukhis, Cooper, & Curt Noll, 2019), where programmers try to express simple programs such as the famous "Hello World" in the most difficult-to-understand ways possible (Mateas & Montfort, 2005).

Dual channel constraints enable the controlled study of predictability in code. In particular, we focus our study on *code expressions*, basic units of code that can be evaluated to obtain a value (e.g., a + b where a and b are variables). Expressions can have *equivalent meaning* in the AL channel of code, but be written in different ways—the AL channel constrains how the NL channel is written. Code is highly expressive; alternative forms abound even in very simple tasks. A wealth of equivalent alternatives exist even for a common iteration trope: `for (i = 0; i < n; i++) {...}`. One could pick a name other than `i` or `n`, flip the conditional (`n > i`), use a different incrementing form (`i = i + 1`), or even potentially start and end the loop differently, *all without changing the meaning*. Indeed, a literal infinity of equivalent forms is possible! Still, programmers dutifully repeat such tropes. Why?

One possible answer is that more predictable code may be preferred by developers if it is easier to produce and easier to understand. In that case, we would expect to see that between two equivalent expression forms, developers will tend to prefer the *less surprising, more predictable form.*

We wish to stress that it is not our goal in this paper to strongly disambiguate between two potential sources for preferring predictability: Programmers may produce predictable code because it is easier for themselves to write, *and/or* because it makes it easier to for other programmers to read—that is, an argument from an audience design perspective (Bock, 1987; Clark & Murphy, 1982; Ferreira & Dell, 2000). In practice, it has been extremely difficult to disambiguate these pressures in a natural language context (Snedeker & Trueswell, 2003; Zhan & Levy, 2018). Our main goal in this work is simply to establish that programmers *have a sense of predictability in code* by demonstrating this exists even at the level of code expressions. Specifically, we wish to see how predictability, operationalized as *surprisal*, is captured in the preferences for expressions by programmers—both in the existence of mature code corpora and in the initial preferences of programmers.

1.3. Overall experimental summary

We aim to test the hypothesis that human preferences for more predictable code are one of the drivers for the observed predictability of code corpora, and more generally test whether humans prefer predictable code, focusing in this paper mainly on *code expression*

predictability. By accounting for the algorithmic channel's effects through meaning-preserving transformations, we can focus on the NL channel, where we theorize predictability/surprisal should behave similarly to natural language. Therefore, we hypothesize that surprisal (generated from a language model) should be *lower* (i.e., *more predictable*) in actual developer-written code expressions found in online corpora, when comparing these "real" code expressions to semantically equivalent alternatives created via meaning-preserving transformations. Likewise, when humans are presented with semantically equivalent expressions, we hypothesize that they will prefer the more predictable fragment.

To explore the relationship between programmer preference and surprisal, we adopt a *triangulation* approach, combining a *corpus experiment* with two subsequent *human subject* studies. A large code corpus embodies numerous choices made by programmers and thus is a representative sample of these choices. Within this corpus, we can examine the occurrence frequency of different implementation choices of the same computation, and determine if particular choices dominate. To preview our findings, we find that language models generally score expressions originally written by developers as *more predictable* (*lower surprisal*) than expressions produced from meaning-equivalent transformations, though the effect varies across transformation types. Then, to better understand how surprisal in code relates to human preferences, we perform two forced-choice surveys with programmers recruited online. In these studies, we seek to answer if surprisal captures programmer preferences. Are the more predictable variants of code expressions (according to the language model) also the variants that humans prefer? We find that programmers do exhibit preferences for equivalent, but less surprising code expressions, and that more powerful language models like the transformer more frequently and consistently model these preferences. At a high level, we would wish for this paper to help serve as an early step in providing a more grounded psycholinguistic perspective of code as language, such as those called for by Fedorenko, Ivanova, Dhamala, and Bers (2019), which we expand upon with a road map in our discussion. All our data, R notebooks, and results for each of these experiments can be found at https://doi.org/10.5281/zenodo.2573389.

The rest of the paper is structured as follows. First, Section 2 provides some background of existing research in program understanding. Then, Section 3 describes our corpus study. Section 4 describes our experiment looking at extreme differences in surprisal, and Section 5 describes our experiment looking at gradient differences in surprisal. Finally, Section 6 discusses our results, placing them in the context of existing knowledge, and suggesting how this work can be built upon in future work.

2. Background

Program comprehension research is decades old, and earlier models are being reexamined in light of newer brain and eye-tracking techniques. Questions on what measures of comprehension to use and how to validate them are of great interest. (Siegmund [2016] provides a helpful survey of the field.] Surprisal has not seen as much focus as it has in

natural language, so we highlight some existing research and relate it to surprisal when possible.

Early studies of program understanding used methods such as "think-aloud" paradigms, in which programmers narrate their thoughts while doing tasks like debugging, to begin probing the cognitive processes that play a role in coding. A major contribution of these paradigms is to demonstrate the extensive use of top-down (Brooks, 1978, 1983; Von Mayrhauser & Vans, 1993) as well as bottom-up (Pennington, 1987; Shneiderman & Mayer, 1979) knowledge in programming. Rather than simply constructing their understanding of a program from bottom-up reading and analysis, developers make hypotheses about a program's functionality that they iteratively reevaluate as they read and understand the code (Siegmund, 2016). For example, a function named "binarySearch" could provide an obvious cue to a programmer that the function implements the classic search algorithm that quickly finds an element in a sorted list. Early theories focused on these so-called *beacons*, cues to meaning that often arise from the natural language elements of code. Von Mayrhauser and Vans (1993) highlight developers' ability to switch between considering different hierarchical levels of code (e.g., considering the many interacting components of code versus considering specific lines or sections). Shaft and Vessey (1995) demonstrate that developers can rely more on top-down hypothesizing when working on programs from a familiar (compared to an unfamiliar) domain.

There is some literature suggesting that code that conforms to expectations is easier to understand. Cognitively, studies have found that "regular code"—code patterns that are repeated in succession—are less cognitively difficult than versions without those repetitions (Jbara & Feitelson, 2014, 2017). Siegmund et al. (2017) found that comprehension is more efficient (requiring less neural activation) when semantic cues that facilitate top-down comprehension are available, compared to when they are obfuscated (e.g., comparing meaningful function identifiers like "binarySearch" to random letter strings).

Indeed, obfuscation has long been a technique used to change code to foil reverse-engineering, making it so that the intended meaning is difficult to recover, either automatically or by human effort. Program obfuscation involves transformations that largely preserve the meaning of the algorithmic (AL) channel, but confuse the natural language (NL) channel. It can be used for protecting intellectual property or to maliciously disguise harmful code to evade detection. The transformations are approximate, because obfuscation can allow the program to produce different error behavior and run much slower (Collberg, Thomborson, & Low, 1997, 1998). Predictability has also recently been used to both aid and reverse obfuscation techniques. It has been used to determine what combinations of obfuscating transforms should be applied, though this was driven more by a desire to make obfuscations difficult for machine learners to reverse than human readers (Liu et al., 2017). In the opposite direction, language models have been used to recover obfuscated variable names in JavaScript, treating the task as a translation problem to recover the real variables from minified ones such as "a," "b," "c," etc. (Vasilescu, Casalnuovo, & Devanbu, 2017). As such, we have good reason to suspect that predictability should relate to human preferences for, and understanding of, source code.

Moreover, in programming languages as in natural languages, people demonstrate preferences for specific templated patterns of production, such as formulaic language or *idioms*, which can quickly and efficiently convey meaning for those familiar with them, for example, `for(int i = 0; i < len; i++)` (Schmitt & Carter, 2004). Allamanis and Sutton (2014) were able to find and extract idioms from code, demonstrating that they appeared for a wide variety of contexts and tended to be presented as learning examples on StackOverflow, a question-and-answer resource site for programmers. Another common source of templated patterns in code is Application Programming Interface (API) usage. APIs are groups of related functions that give application developers access to common, shared services; the Android API (Google, 2020) is a popular one. APIs are used in idiomatic patterns, (e.g., opening and closing files or accessing data from a service), which, if captured, can teach developers how to use the API, or can be checked for incorrect, energy inefficient, or even malicious usage (Linares-Vásquez et al., 2014; Sami et al., 2010; Zhong, Xie, Zhang, Pei, & Mei, 2009). On a larger scale, developers employ *design patterns*, which are solutions to common problems in software design, and can be used to quickly signal the structure of the code to others familiar with said patterns (Gamma, 1995). Likewise, projects often make use of style guidelines to format code, and research has shown that contributions that are more predictable in the local context of a specific project are more likely to be accepted by that project's community (Hellendoorn, Devanbu, & Bacchelli, 2015).

More recently, methods such as eye-tracking and neuroimaging have been applied to the question of program understanding (Siegmund, 2016). Functional magnetic resonance imaging has been used to compare brain regions used in comprehending code to those used in natural language processing, with mixed results: Some studies (Castelhano et al., 2019; Duraes, Madeira, Castelhano, Duarte, & Branco, 2016; Floyd, Santander, & Weimer, 2017; Siegmund et al., 2014, 2017) report overlap in these brain regions, while Ivanova et al. (2020) report minimal overlap. Eye-tracking studies (see the survey by Obaidellah, Al Haek, & Cheng, 2018) also reveal differences between how humans understand code versus natural language. For instance, studies show that while natural language reading largely follows a linear reading pattern (left to right in English), code readers jump around quite a bit (e.g., from a variable or function use to its declaration; Busjahn et al., 2015; Jbara & Feitelson, 2017). Moreover, expert code readers tend to show more nonlinear eye traces than novices (Jbara & Feitelson, 2017).

In summary, the previous literature on program understanding reveals some high-level similarities to natural language comprehension in the use of predictability, particularly with respect to idioms and top-down expectations about program structure. However, it also reveals differences in terms of eye movement patterns and potentially neural regions. The previous literature has not focused on the relationship between predictability and programmer preferences/ease of processing at the level of individual tokens or lines of code. Since the influence of predictability on the processing of individual words (or other highly local structures) in natural language is among the most foundational results in psycholinguistics, we see establishing a comparable result in program understanding as key

to establishing the cognitive similarities and differences between programming and natural language processing.

3. Corpus study

The first part of our work is a corpus study examining how programmer preferences relate to predictability in naturally occurring code. Our logic is as follows: To the algorithmic (AL) channel, it does not matter whether the programmer writes `i + 1` or `1 + i`. But `i + 1` is more predictable (whether measured by human intuitions or a language model). Thus, if we consistently see more instances of `i + 1` (across different programmers, different projects, etc.), it is evidence that programmers prefer to write the more predictable form (in their mature, edited code). While this preference may seem obvious in this simple example, it is much less obvious that programmers would have a consistent preference (again, across different people, different projects, etc.) about more complex expressions such as byte `eventZone = message[2 + 2 * i];` vs byte `eventZone = message[I * 2 + 2];` (93.5% preferred the second way) or about the use of parentheses in expressions such as `Integer minute = val – (hour * 60);` vs `Integer minute = val – hour * 60;` (90% preferred the first way).

Since these complex expressions may occur only once, we cannot simply count instances of `byte eventZone = message[2 + 2 * i];` vs `byte eventZone = message[i * 2 + 2];` to determine which one programmers are more likely to write. Instead, we take existing lines of code and construct alternatives that use meaning-equivalent expressions. We then measure the predictability of both the original code expression and the meaning-equivalent alternative using language models. If programmers indeed preferentially write more predictable code (even in these potentially complex and novel expressions), then *these code expressions as originally written* should be less surprising to language models than the constructed meaning-equivalent alternatives.

As our primary programming language of study, we choose Java, as it is widely used and a common baseline for studies of language models in code (Casalnuovo et al., 2019; Hellendoorn & Devanbu, 2017; Hindle et al., 2012; Tu, Su, & Devanbu, 2014). In order to provide some more nuance and robustness to our results, we wish to validate our results across different kinds of language models and different levels of abstractness to the models, and we also performed a smaller validation study in another programming language. Therefore, for our corpus study, we break down this more general question of predictability in code expressions into four research questions. First, however, we provide a slightly more in-depth description of surprisal in the context of language models and give a brief summary of the kinds of meaning-preserving transformations we perform.

Language models typically assign probabilities contextually, for example, the probability of an utterance U occurring in context C: $p(U|C)$. The more likely U is in context C, the higher the probability. These models (e.g., n-gram models, recurrent neural networks,

transformers, etc.) are trained on a large corpus of text, and then are evaluated by scoring against a separate, held-out *test* corpus of code that was *not used* in training that model. Performance is measured from the average surprisal on the test corpus. A good model, presented with typical text, will find it highly probable, and thus it will score a low surprisal for most utterances (tokens) in the text and have overall low entropy. Modern language models, well-trained over a large, diverse corpus, can reliably[1] capture the frequency of occurrence of textual elements in that corpus, thus capturing the preferences of programmers who created it. If these preferences are skewed enough toward specific implementations, then *even in the context of new unseen projects* we would expect the code expressions developers choose to write will be more predictable than alternative equivalent implementations. More details of the specific language models used can be found in Section 3.2.

Meaning-preserving transformations can capture a range of possible implementations for a given meaning. While many are possible, we look at small expression-level rewrites. These rewrites include equivalent reordering of the operands in an expression, adding and removing nonessential parentheses, and variable name shuffling (for details, see Section 3.3). These transformations are performed on code fragments from a corpus "unseen" by the language model, which is then used to score the surprisal of the original and the transformed versions. Then, given these transformations, our first research question is:

RQ1. *Given a trained language model, when we perform meaning-preserving transformations on previously unseen test Java code expressions, to what degree does the language model find the transformed code more improbable (higher surprisal) relative to the original?*

Next, to ensure that this is not simply an effect of the choice of programming language, we also choose a secondary language that is different from Java, Python. Python is quite different to Java, so if we see similar effects it provides some evidence that they may be more general than Java. So we ask:

RQ2. *Is language model preference for the original code expressions also observable in Python?*

We would also like to explore how local variances in style (locality) affect the consistency of choice. By locality, we mean that source code files tend to establish their own local vocabulary and style *specific to that file*. Source code has strong locality, much more so than natural language (Hellendoorn & Devanbu, 2017; Tu et al., 2014); cache-based language models capture this local context by storing recently seen patterns in the text in a refillable *cache*. For example, while `i = 1 + i;` is an unusual way to write an expression, if a developer consistently writes that way in a file, a cache model will find it less surprising as it processes more of the file. Likewise, a developer making the unusual choice to use "iteratorVariable" instead of the common "I" would also become less surprising in the local context of the file.

Additionally, we would like to consider if these preference patterns are retained in the underlying structure of the code expression—that is, when identifiers (names of code variables like i or n) and literals (numbers like 0, 1, 2.5 and strings like "Hello World" are all examples of literals) are abstracted. So, for instance, while `i = i + 1;` is a preferred pattern over its alternative form `i = 1 + i`, by abstracting the expression, the model could learn if developers tend to write `integer_variable = integer_variable + integer_literal;` over `integer_variable = integer_literal + integer_variable;`. If the transformations still have higher surprisal after this abstraction, it provides evidence for preferences over the underlying patterns, rather than being tied to specific variable names.

Thus, to investigate what occurs when varying the model to account for locality and abstractness, we ask:

RQ3. *Do cache-based models that incorporate local style discriminate between the original and transformed expressions more strongly? Is the preference for the original retained even when abstracting identifiers and literals?*

We expect that some transformations will disrupt the expression less or potentially even make it more probable to our models. In particular, we would like to see how the original surprisal of the expression relates to the effect of the transformation. We would expect highly improbable expressions to have greater potential to become more "typical" after transformation. Such code expressions might be associated with fewer restrictions on developer choice. Thus, we ask:

RQ4. *How does the surprisal of the original code expression relate to the effect of the transformation? Do high-surprisal and low-surprisal code behave differently?*

With these questions in mind, we will now describe our data selection and filtering process, language model setup and parameters, and our choice and validation of meaning-preserving transformations.

3.1. Data

Our experimental dataset is chosen to help control for potential confounds, while also affording enough opportunities for transformations. Since our main focus is Java, (see RQ1) we use a larger Java corpus and replicate with a smaller corpus in Python (RQ2).

We cloned the top 1,000 most starred projects on GitHub (GitHub & Inc., 2020) for Java and Python. GitHub is the most popular host of open source code projects on the web. Stars are a measure of the popularity of the project, which ensures that we are drawing code from prominent projects. We use a subsample of these projects due to computational constraints by selecting the 30 projects from Java and Python with the most locations for possible transformations. Specifically, we put all the projects through the first step in our process which identifies possible transformations via the Abstract Syntax

Tree (AST) and counted locations where a meaning-preserving transformation could be made. Manual review of these projects in Java and Python showed nearly all of them were actively maintained (all but one had updated in the past few months), and represented a diverse set of application domains. These projects are then randomly divided by project into a 70–30 training/test split.

Duplication (often called "code cloning") can be a potentially confounding effect in training and testing code with language models (Allamanis, 2019). Since our focus is on programmer preferences for certain coding forms, it would be inappropriate to remove all clones, that is, code that developers reuse by copying and pasting it elsewhere (possibly with some modification). Though some consider cloning to be bad practice, there are valid reasons for developers to use code clones, such as language limitations or because they provide a template of code developers wish to reuse (Kim, Bergman, Lau, & Notkin, 2004). These are exactly the kinds of patterns that developers use that we hope our language models will capture, so removing them does not make sense. Still, duplication at the level of entire files may be the result of standard project files that are automatically generated by tools, or even just a developer copying a useful module between projects. In those cases, it is unlikely developers examine the code as closely as when they are copying a template that might need to be altered to fit a new context, and so we believe that our language models should exclude them, too. Therefore, we do a lightweight removal of fully duplicated files. This lightweight process compares the name of every file and its parent directory (e.g., main/ExampleFile.java), keeping the first one seen of an equivalent set, and removes the others from our training/test data.

Table 1 shows the file-counts and approximate token-counts in the training set for each corpus. Duplication filtering removes 6.1% and 10.7% of files in Java and Python respectively. Despite sampling the same number of projects, the Java corpus is much larger, but Java is our main focus. Since we use Python to check if the results replicate to another language, the smaller Python dataset is not a major concern.

Our test data were chosen to be **distinct** from the training data. In addition to the file level filtering used in both the training and test sets, when testing we filter out lines commonly associated with generated code, coming from *equals* and *hashCode* functions. We dropped lines with the strings "hashCode" or "other" which we observed manually to be contributing to this repetitiveness. In the case of our identifier shuffling transformations that operate at the method rather than expression-level (see Section 3.3), we instead removed all equals and hashCode methods. These lines are generated by Integrated

Table 1
Summary of Java and Python dataset sizes, in terms of total files, files after applying our filters to remove potentially copy-pasted code, and input tokens during language model training

Language	Files	Unique Files	Training Tokens
Java	204,489	184,093	~118.5 M
Python	27,315	23,105	~18.2 M

Development Environments (IDEs),[2] and they arguably do not accurately represent *human written choices* (or at least, style choices so codified that they have been automated). We also remove from the test data identical lines of code appearing more than 100 times (a threshold of 10 gave similar results, suggesting robustness), as these may also be at risk of extensive copy-pasting. As mentioned previously, we believe that it would *not* be correct to simply filter out all duplicated expressions, as it is perfectly valid for developers to rewrite the same code.

Since our study operates at the primarily expression level, it is difficult to precisely find and account for copy-pasting; we argue our approach gives a reasonable middle-ground between removing the extreme cases while still retaining most of the natural repetition of code. Finally, we note that we *did not* remove repeated or generated code *fragments* from the *training* data to properly reflect the code that programmers would read (and learn preferences from). Our test set pruning was to avoid overly weighting repeated and generated code, and emphasize more the individual, independent choices made when *writing* code.

3.2. Language models

We estimate a language model *LM* over a large corpus, and then use the $P_{LM}(C)$ from the language model as an indication of predictability. Specifically, we use the surprisal of the language model with respect to a fragment C, which is precisely $-\log(P_{LM}(C))$. Lower surprisal indicates code is more predictable based on the patterns learned from the training data.

We use four *n*-gram language model variants to capture various aspects of possible developer preference. First, we use a basic 6-gram model with Jelinek-Mercer smoothing as recommended previously (Hellendoorn & Devanbu, 2017), denoted as the *global* model. To answer the two parts of RQ3, we first use Tu et al.'s (2014) *n-gram-cache* (henceforth abbreviated as the *cache* model) to capture local patterns. Then, we build an alternate training and testing corpus where we use the Pygments (Brandl & Chajdas, 2020) syntax highlighter to replace all identifiers and types with generic token types, and literals with a simplified type. For numerical literals we keep integers like 1,2,3 and replace larger integers and floating point numbers with labels like <int> and <float>, and for strings we keep the empty string, single-character strings, and replace everything else with <str>. Table 2 shows an example along with the Pygments' internal types on a simple line of code. These models are implemented in the SLP-Core framework by Hellendoorn and Devanbu (2017) and Hellendoorn (2017). To assess preference, we compare the average surprisal of tokens that appear *only in both the original and the transformed version of the expression*. The tokens not involved in the changed expression are not considered.

Finally, we validate the robustness of our *n*-gram results with two neural models. First, we considered a small 1 Layer LSTM implemented in TensorFlow, trained with 10 epochs and 0.5 dropout. Second, we use a *transformer model* (Vaswani et al., 2017) combined with modified byte pair encoding (BPE; Gage, 1994; Sennrich, Haddow, & Birch,

Table 2
A representation of a line of code, its Pygments representation, and the abstracted version

Original Code	int	start	=	index	+	1	;
Pygments Internal Representation	Keyword. Type	Name	Operator	Name	Operator	Literal. Number. Integer	Operator
Abstracted Code	int	Token_Name	=	Token_Name	+	1	;

"Names" are abstracted (which Pygments uses to represent identifiers and more complex types) along with some literals. So here we see the primitive type "int" and the simple literal "1" are left concrete, but the identifiers are not.

2016). BPE has been shown to be effective in managing the problem of code's large vocabulary (Karampatsis, Babii, Robbes, Sutton, & Janes, 2020) by dividing the tokens into subtokens with similar frequency, and transformer models are considered to be much more powerful than *n*-gram models. We trained/tested on the same projects as the *n*-gram models after applying BPE for 10 epochs, using a two-layer transformer, with eight heads, attentional and hidden dimensions of 512, a learning rate of 0.2, a dropout rate of 0.1, batch size of 15,000, and 200 subtokens per sequence. The transformer was masked so that it used only previous tokens as context, making this context comparable to those used by the *n*-gram and LSTM models. While transformers have not seen as much exploration in a source code context as they have in natural language, they have been applied in variable misuse tasks (Hellendoorn, Sutton, Singh, Maniatis, & Bieber, 2020) and have been used recently in industrial code completion (Svyatkovskiy, Deng, Fu, & Sundaresan, 2020). While the specific configuration details are not essential to our experiments, we include them to aid in replication.

Both of these neural models are evaluated on the subset of transformations that we use in both the corpus and the human subject study (the Java swapping and parentheses transformations; see Section 4.1 for more details), which excludes the variable renaming transformations. The renaming transformations were more computationally expensive and were intended more for validation (see Section 3.3). For the small-scope localized transforms we use, we believe the *n*-gram models are all adequate to measure meaningful properties of the corpus, as such models have been used in prior work of this nature (Casalnuovo et al., 2019).

3.3. Meaning-preserving transformations

We choose not to use existing code transformation tools, as they are either not meaning-preserving (e.g., mutation testing tools; Just, 2014; Madeyski & Radyk, 2010)[3] or operate at the wrong scale of code object, such as compiler optimizations. Compiler optimizations are designed to retain program behavior, but most do not operate at the source code level, and often restructure code quite dramatically, resulting in unfamiliar constructions. While it seems highly likely that these broad restructurings would affect programmers' preferences and comprehension, a more stringent test of preferences is to predict

transformations to expressions within a *single line* of code, such as swapping the order of operands.

Thus, to provide the strongest test of our hypotheses, we use source-level transformations that both are meaning-preserving *and* local in scope. Our focus is primarily on transforming source code *expressions*, done by manipulating Java and Python *ASTs*. ASTs are representations of program syntax, in tree form, used by compilers and IDEs to analyze and edit program syntax. They resemble constituency trees in natural language, though in code, they are called *abstract* as they omit some details of syntax; for example, punctuations, like the ";" that end Java statements, are elided. We use the *AST* Parser from the Eclipse Java development tools (JDT; Eclipse Foundation, 2020a) and the *ast* module from Python 3.7 (Python Software Foundation, 2020).

We implement 12 different transformations (see Table 3) grouped roughly into three categories: (a) swapping transformations, (b) parentheses transformations, and (c) renaming transformations. The second column of Table 4 gives a breakdown of the combinations of language models and transformations used in the corpus study. There are six nonoverlapping subgroups: arithmetic and relational swaps, parentheses adding and removing, and shuffling identifiers within and between types. We implement all of these transformations in Java. However, in Python, as types are not enforced and the language tools are more limited, we replicate only one type of transformation, swaps over relational operators. These limitations also required us to normalize the original Python files with *astor* (Peksag & Maupin, 2019), which can slightly change parentheses. As we do not perform parentheses transformations for Python, this should only minimally impact results.

Swapping transformations: We have eight kinds of transformations involving swapping and inverting operators, divided into two subcategories. The first subcategory swaps arithmetic operands around the commutative operators of + and *. We swapped *very* conservatively. We limit the types of the variables and literals in the expression to doubles, floats, ints, and longs. Infix expressions with more than two operands are only transformed if the data type of the operands is int or long to avoid accuracy errors due to floating-point precision limitations. We also exclude expressions that contain function calls, since these could have side effects that alter the other variables evaluated in the expression.

Table 3
Pseudocode examples for the transformations

Swapping operands	Arithmetic	*	a * b	b * a
		+	a + b	b + a
	Relational	==, !=	a != b	b != a
		<, <=, >, >=	a <= b	b >= a
Parentheses manipulation	Adding		a + b * c	a + (b * c)
	Removing		a + (b * c)	a + b * c
Shuffling variable names	Within variable types		int a int b	int b int a
	Between variable types		int a float b	int b float a

Table 4
Summary of which language models were applied to which transformations in each study

Language Model	Corpus Study	Human Subject Study
Global Ngram	Swaps (Java/Python), Parentheses, Variable Shuffling	Swaps (Java), Parentheses
Cache Ngram	Swaps (Java/Python), Parentheses, Variable Shuffling	—
Global Abstract Ngram	Swaps (Java/Python), Parentheses	—
Cache Abstract Ngram	Swaps (Java/Python), Parentheses	—
LSTM	Swaps (Java), Parentheses	—
Transformer	Swaps (Java), Parentheses	Swaps (Java), Parentheses

The second subcategory of operator swapping involves the six relational operators, ==, !=, <, <=, >, >=. We flip the subexpressions that make up the operands for each of these, either retaining the operator if it is symmetric !=, ==, or inverting it if it is asymmetric (e.g., > becomes <). While we do not limit the types in these expressions as they are commutative and do not risk changing floating-point behavior, expressions with function calls are excluded to avoid side effects.

Parentheses transformations: The next category involves the manipulation of extraneous parentheses in source code. Programming languages have well-defined operator precedence, but programmers can still (and often do) include extraneous parentheses, possibly to aid in readability or simply out of habit. For instance, in cases where less common operators are used (such as bit shifts), the parentheses may make comprehension easier, leading to a preferred style.

Therefore, we can transform expressions by adding or removing extraneous parentheses. The parentheses adding transformation relies on the tree structure of the AST to insert parentheses while preserving correctness in the order of operations. Parentheses are not added to expressions whose parent is a parenthesized expression to avoid creating double parentheses. For example, we would not change `(a * b) + c` to `((a * b)) + c`. Parentheses are also never added around the entire expression.

For parentheses removal, we select each parenthesized expression. Then, each of these is passed to the `Necessary-ParenthesesChecker` from the Eclipse JDT Language Server (Eclipse Foundation, 2020b) to check if removing them would violate the order of operations. Any subexpressions that pass this check are then considered candidates for removal. This method is used by the same algorithm supporting the "Clean Up" feature within the Eclipse IDE.[4]

Variable shuffling transformations: Finally, we consider transformations that shuffle the names of identifiers. To avoid changing meaning, we swap only within a method, using the key bindings of the *AST* to maintain scoping rules, which define how variables with identical names but different scopes are resolved. Scopes define the regions in code in which a variable can be referred to. For example, if a variable defined inside an if statement has the same name as one outside it, references using this name refer to the one inside the if statement. It has a more "local scope" and thus

has priority. If a variable name is used for a declaration more than once in a function (e.g., multiple loops using `i` as a variable for iteration), it is excluded to avoid assigning two variables the same name within the same scope. Methods containing lambda expressions are also ignored because their variable bindings are not available in the *AST*.

We separately consider renaming both *within types* and *between types*. As an example, consider a function with two *int* and two *String* variables. In the *within types* case, we only consider replacing one integer's name with the other, and the same for the strings. In the *between types* case, all four variable names can be assigned to any of the integers or strings other than their original variable. We expect that names given to the same types will be used more similarly than names given to different types. Thus, we would expect transformations *between types* would result in code relatively more improbable than those produced by transformations *within types*.

We note that unlike the prior two transformation groups, our variable renaming transformations act as more of a check on the validity of our experiment. It has been long known that having poorly named or misleading variables makes code more difficult to read, and this is a commonly exploited basic technique to obfuscate the meaning of code (Collberg, Thomborson, & Low, 1997). Therefore, altering developer-chosen variable names should create confusing code, and the language models should strongly prefer the originals.

In contrast, rules around the ordering of operands and usage of parentheses are not as clear cut, and comparing the surprisal of the transformations can inform us of how relatively strong convention is among them. To confirm this, we searched the projects in our training and test sets for references to style guides. If common style guides prescribe operand ordering and parentheses behavior, this could influence our results. We found several style guides for both the Python and Java projects. Some projects even had explicit automated style checks, while others had much looser guidelines (for instance, Apache Tomcat). Virtually all the guidelines were largely unrelated to our primary transformations of swapping operands and the use of clarifying parentheses, and more focused on naming and whitespace conventions. We did find a few, limited references in Java to using parentheses as needed for clarity; and just one project specified that *null* values should come second (e.g., `x == null` instead of `null == x`, though we did find developer-written examples of the opposite order), but otherwise nothing that would affect our transforms.

Finally, we acknowledge that patterns of variable names use in particular might be learned more strongly than the swapping or parentheses transformations, particularly if one domain of application was severely overrepresented. Though we manually examined all the projects in both training and test sets and found a wide variety of projects, we note that in Java several of the projects tended toward database applications and in Python toward machine learning. While we believe the existing diversity or projects combined with our duplicate filtering helps minimize this factor, we acknowledge this as a possible influence in the strength of patterns learned.

3.3.1. Selecting transformation samples

As expressions grow in size, the number of possible transformations grows exponentially. Generating all these transformations is neither feasible nor desirable, so we select a random subsample. For the operand swapping and parentheses modification cases, we randomly sample up to n transformations, where n is the number of possible locations to transform in the expression. For variable renaming, we consider only functions with up to 10 local variables that can be shuffled.

Though our transformations are sound by design (i.e., they preserve meaning on the AL channel), we also hand-checked a large sample of the results to ensure correctness and diminish the possibility of error. We did not find any examples of meaning being changed. There were rare examples of cases with double parentheses, which would be highly unlikely for humans to produce. Also, as the transformations are randomly created, some of them might be very unusual to humans, such as converting `(a == null) || (b == null)` to `a == null || (b == null)`. In the human subject study, we handle these and similar cases carefully (see Section 4.1 for details).

3.4. Modeling

To compare surprisal before and after the transformation, we use paired nonparametric Wilcox tests (Hollander, Wolfe, & Chicken, 2013) and associated 95% confidence intervals, which measure the expected difference in medians between the original and transformed code expressions. We also *widen* the intervals using the conservative family-wise Bonferroni (Weisstein, 2004) adjustment, to account for the tests on each model and transformation.

To answer RQ4, we turn to regression modeling. Recall that we theorize that expressions that are *more improbable* to the language models should be more amenable to becoming more probable after transformation, whereas *low surprisal* would be associated with stronger preferences and thus transformations are more likely to be harmful. We measure this effect with ordinary linear regression, predicting change in surprisal (from the transformation) as a function of original surprisal. We additionally include controls for the length of the line, the type of *AST* node that is the parent of the expression, and a summary of the operators involved in the expression. In the case of multiple operators, we selected the most common one from the training projects to represent the expression. Our regressions are limited to single transformations for ease of interpretation, and we filtered out rare parent and child types (<100).[5] We identified influential outliers using Cook's method (Cook & Weisberg, 1982) and removed those with values greater than $4/n$. We examined residual diagnostic plots for violations of model assumptions, and made sure multicollinearity was not an issue by checking that variance inflation (VIF) scores were <5 (Cohen, Cohen, West, & Aiken, 2003).

3.5. Corpus study results

3.5.1. Swapping expressions

We have 20,829 instances of transformable arithmetic swaps in our Java data, and 133,845 and 32,219 instances for relational swaps in Java and Python respectively. Fig. 1 shows the difference in n -gram surprisal (transformed − original) for all of the swap transformations, with Fig. 1a showing the Java arithmetic swaps, Fig. 1b showing the Java relational swaps, and Fig. 1c showing the corresponding Python relational swaps. Rows 1, 2, and 3 in Table 5 show the associated Wilcox tests and confidence intervals around the median for each of these model variants.

The data for the swaps support our theory that the language model would rate the original (unaltered) form more probable to varying degrees (even though both forms mean

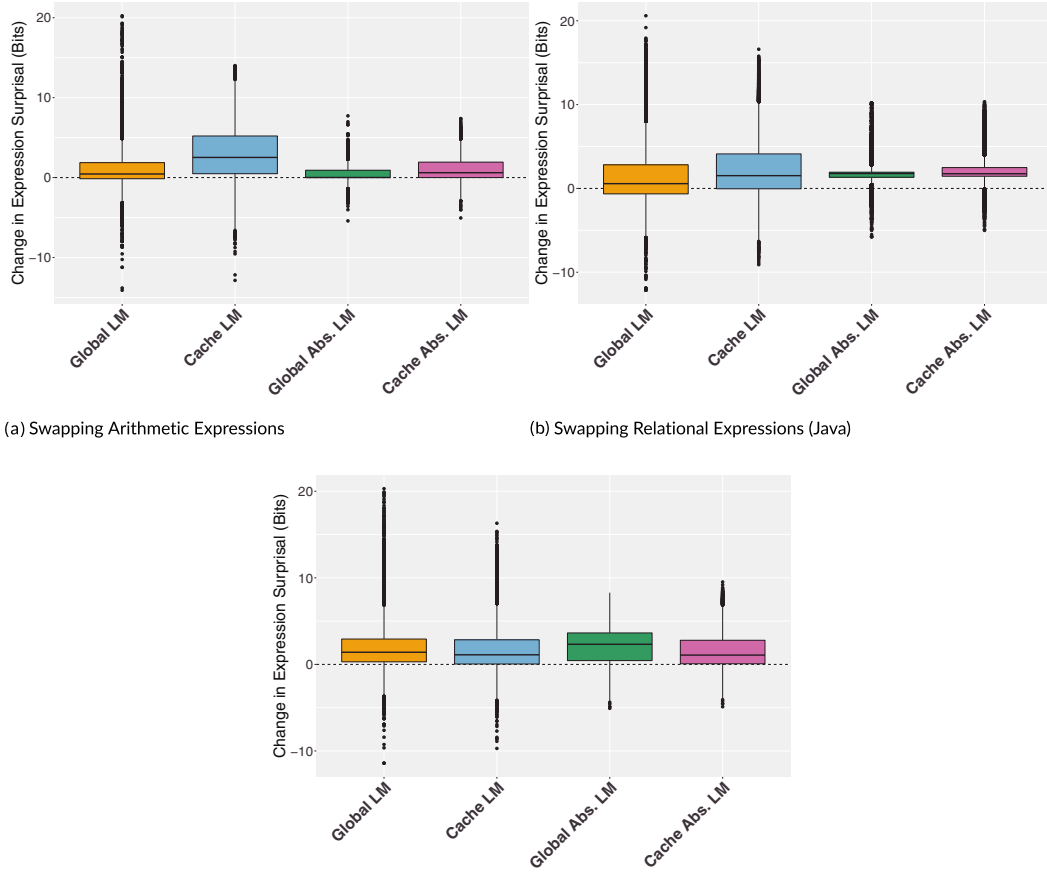

Fig. 1. Average surprisal change after arithmetic and logical swap transformations. Positive values indicate an increase in surprisal.

Table 5
Two-sided paired Wilcox signed-rank tests and 95% confidence intervals of surprisal difference *original source minus transformed source*

	Global	Cache	Global Type	Cache Type
Arithmetic swap	−0.7972, −0.7045	−2.9691, −2.8241	−0.5578, −0.5053	−1.4687, −1.3897
Relational swap (Java)	−1.2171, −1.168	−1.964, −1.905	−1.7204, −1.7009	−1.954, −1.936
Relational swap (Python)	−1.7643, −1.6837	−1.4616, −1.3778	−2.4814, −2.4061	−1.5448, −1.474
Add parentheses	−0.3344, −0.3201	−0.5771, −0.5432	−0.2351, −0.2237	−0.3783, −0.364
Remove parentheses	−0.0106, 0.0346○	−0.4393, −0.3605	−0.2007, −0.1494	−0.4233, −0.3626
Variable shuffle (within types)	−0.807, −0.6767	−0.9744, −0.8402	—	—
Variable shuffle (between types)	−1.7318, −1.6114	−2.3273, −2.2308	—	—

A 1-bit negative difference indicates the original code is twice as probable as the transformed one. Intervals are Bonferroni corrected. ○indicates $p > .05$, otherwise $p \ll .001$.

the same thing). To focus on one transformation and model, in the Java arithmetic swaps, the global language model finds the original expression 1.68 times more probable (0.75 bits of surprisal less) than the transformed version, and the cache model finds the original 8 times more probable (3 bits less) than the transformed version (the increase can be measured as 2^{diff} e.g., $2^{0.75} = 1.68$). The cache language model does discriminate better across all the Java transformations, both with concrete and abstracted identifiers; but *not*, however, in Python, perhaps due to a global cross-project Python community-specific culture that might be stronger than in Java. For the models of the identifier abstracted code expressions, the original expression is once again more probable, although the cache effects mirror their behavior in Java (where the transformed expression is even more surprising) and Python (where it is less surprising). Importantly, the fact that the original code expression is rated more probable by the language model suggests that there are structural frequencies learned from the training data on top of the more clear cues for names.

Rows 1 and 2 of Table 6 show the LSTM and transformer models we ran to validate the robustness of the *n*-gram models on Java swaps.[6] They both also rate the original code more probable. The LSTM suggests a stronger effect with relational swaps, whereas the transformer shows this for the arithmetic swaps. Therefore, we cannot clearly say in which the effect might be stronger.

Now, to answer RQ4, we will describe one regression model in-depth: the one for the global arithmetic swaps. We present only one of these models in detail as the results were consistent across transformations and language models; the only exception occurred in Python and is described after this example.

Table 6
Two-sided paired Wilcox signed-rank tests and 95% confidence intervals of surprisal difference *original source minus transformed source*

	LSTM	Transformer
Arithmetic swap	−0.7517, −0.6264	−1.098, −1.0066
Relational swap (Java)	−1.0063, −0.9516	−0.479, −0.4621
Add parentheses	−0.5433, −0.4919	−0.2021, −0.1698
Remove parentheses	−0.5559, −0.4247	−0.0591, 0.01○

A 1-bit negative difference indicates the original code is twice as probable as the transformed one. Note that the LSTM values show the difference between averaged shared surprisal at the *token* level, whereas for the transformer they are at the *subtoken* level, since we used BPE. Intervals are Bonferroni corrected. ○indicates $p > .05$, otherwise $p \ll .001$.

We model *Surprisal Change ~ Original Surprisal + log(NumTokens) + ParentOperator + Operator*, where the change in surprisal is predicted by the original surprisal, controlled by both the log size of the expression in tokens, and the parent and operator types of the expression. For every bit increase—meaning the expression is twice as difficult for the model to predict—in the original expression, the change decreases by 0.279 bits. This effect is quite strong, explaining nearly 36% of the variance in the difference. We can conclude that when the original variant is less predictable, the effect of a transformation will either lead to a smaller increase or even *decrease* the surprisal of the original expression. This could be explained either by these higher surprisal original expressions being cases where model ratings are overall less clear, *or* possible cases where the original developer inadvertently wrote a less preferred form. This negative correlation between original surprisal and the change also holds in the regressions for all the other language models on this transformation. Among the controls, longer expressions are also more likely to be amenable to transformations, and while most parent nodes in the *AST* are similar to the "==" baseline, return statements and array accesses tend to have less strict style. Finally, swaps that occur on a "+" instead of a "*" are less probable to the language model with an increase of 0.64 bits in the surprisal difference. This may be the result of addition being much more common than multiplication; patterns that are used more frequently may become more fixed into a particular style (for a comparable result from natural language, see Morgan & Levy, 2015).

This effect of higher original surprisal leading to greater opportunity to make code more predictable persists across all the regressions for the Java relational swaps, and the Python swaps on the *concrete* expressions. However, in the abstracted Python expressions, this effect is reversed, with more surprising patterns in the original expressions leading to even more surprising transformations. Nevertheless, this coefficient explains only a small amount of the variance of the change (<2.5%), suggesting it is quite a small effect. This effect is surprising, and a more in-depth study—possibly across more languages or with human subjects—would be necessary to understand this counterintuitive

behavior in the abstracted code. Perhaps it may be a result of these patterns simply being very unusual and hard to predict.

So for *Java* swapping transformations, RQs 1, 3, and 4 are answered affirmatively, albeit to various degrees. All models rate original forms more likely; cache models more strongly discriminate between the original and transformed code, and less probable expressions are associated with smaller increases, or even sometimes reductions in surprisal. Our results for RQ2 provide additional support for the overall theory but suggest complications in the details. The ability of the locality to discriminate may be language-specific, and the relationship between the original expression surprisal and the change in surprisal in the abstracted models (as its direction was different from all other results) may suggest Python programmers exhibit preferences more closely tied to identifiers than structural patterns. Finally, the abstracted models show us that structural patterns differ in how elements are ordered in code expressions.

3.5.2. Parentheses transformations

For parentheses, we have 63,625 additions and 9,717 removals, with the results shown in rows 4 and 5 in Table 5 and plots of the surprisal differences shown for adding parentheses in Fig. 2a and for removing parentheses in Fig. 2b. The results for surprisal change, the effect of the cache, and the regression models relating the original surprisal to the size of the change are similar to those in the swaps, with one major exception. The difference between the original and transformed expressions in the *global* model when *removing parentheses* is *not significant*!

We delve into this unexpected result more closely using examples in Table 7 using a few case studies of the more extreme differences to see if they make sense. These are fairly intuitive; the largest improvement in predictability comes from removing parentheses unnecessary to clarify the order of operations from around a literal denominator. In contrast, a large increase in the difficulty of prediction occurs with rarely used bit shift operators—suggesting that developers may prefer parentheses around rare operations to clarify the order of operations. Like for the Java swaps, the regression models relating original surprisal to the size of the change all exhibit significant negative relationships for the global, cache, and abstracted models; more surprising expressions are more susceptible to transformations that make them more predictable.

The second two rows of Table 6 show the neural results for the LSTM and transformer models on the parentheses models. The LSTM models align with our theory, although the change is smaller relative to the swaps. However, the transformer models reinforce what was seen in the global *n*-gram models—suggesting that the effect for extraneous parentheses is weak.

Thus we answer RQs 1, 3, and 4 affirmatively, except for models of parentheses removal. We speculate that this may be the result of less consistent style around the usage of parentheses, similar to what Gopstein et al. (2017) and Gopstein, Zhou, Frankl, and Cappos (2018) found with bracket usage. We further discuss the influence of confusing code and style guidelines in Section 6.

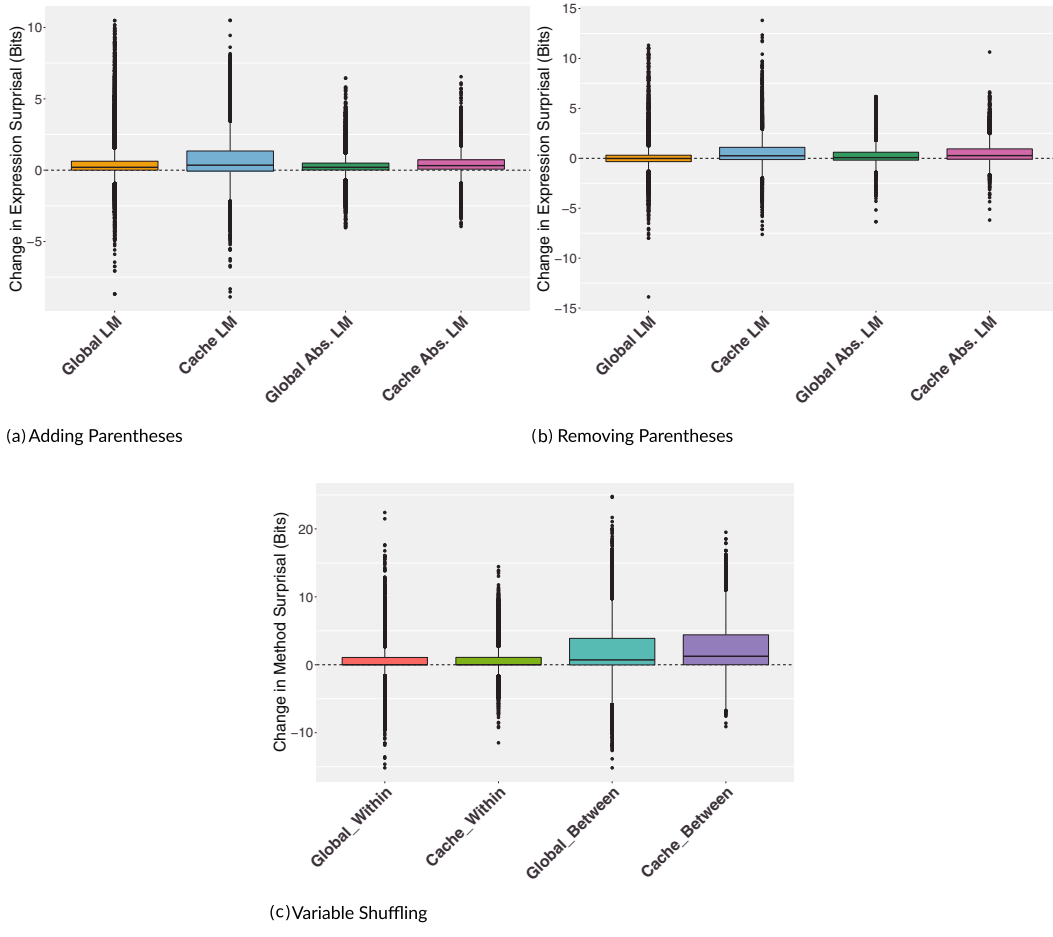

Fig. 2. Surprisal change after adding/removing extraneous parentheses and shuffling variable names. For the variable shuffle, the left two are only within types, and the right two are unconstrained. Positive values indicate an increase in surprisal.

Table 7
Sample meaning-preserving transformations with the largest expression-level surprisal changes (decreases and increases) for parentheses removal with the global model

Direction of Change	Original	Transformed
Decrease	double seconds = time/(1000.0);	double seconds = time/1000.0;
Decrease	return ((dividend + divisor) − 1)/divisor;	return (dividend + divisor − 1)/divisor;
Increase	int elementHash = (int)(element ^ (element>>>32));	int elementHash = (int)(element ^element>>>32);
Increase	c1 |= (c2>>4) & 0x0f;	c1 |= c2>>4 & 0x0f;

3.5.3. Variable renaming transformations

Finally, we consider variable renaming transformations, measuring mean surprisal change across all affected expressions within *the same method*. There are 17,930 methods where we were able to shuffle names within types and 48,160 methods where we were able to shuffle them between types (unconstrained shuffles are possible in more methods), with results in rows 6 and 7 of Table 5 and plotted in Fig. 2c (the abstracted models do not apply as the shuffles operate on the concrete identifier names). As expected, shuffling variable names within a method increases surprisal. Variable names matter for program comprehension, and obscuring these names is one of the most common and simple forms of program obfuscation (Collberg et al., 1997). Moreover, we confirm that swapping variable names across types is more disruptive to predictability; the difference is about twice as large. Cache effects are still present, but diluted, possibly because the shuffle pulls its vocabulary from very similar contexts. As with all other Java transformations, the regression models show that more surprising variable names have less of a surprisal increase after renaming. So in conclusion, the renaming shuffle transformations all answer RQs 1, 3, and 4 as expected, with stronger results when shuffling between rather than within types.

3.5.4. Discussion

Across all the transformations we can see the general trend that we expected. Transformed expressions tend to be scored less likely by the language model, regardless of programming language, with locality-capturing cache models showing a stronger effect in this direction. Our results in our renaming transformations help validate the other results, which show some interesting differences in language model preference strength. While it is not clear which preferences are stronger between arithmetic and relational operator swaps, the preferences there clearly seem to be stronger than around parentheses. Moreover, the transformation for removing unnecessary parentheses is net neutral to global *n*-gram and transformer models, suggesting inconsistent behavior across different projects. When should you include parentheses to help clarify an operation? Our corpus results suggest developers cannot seem to agree! Next, we discuss our human subject studies, which see if a relationship between surprisal and direct human judgments of preference exists and if similar variations in preference strength across transformation type persist.

4. 4. Human study 1: Extreme *n*-gram surprisal

In our corpus study, we saw that language models find transformed (but meaning-equivalent) code expressions less probable (higher surprisal) than the original human-written versions. However, the extent to which the judgments of language models actually align with the preferences of real programmers is not clear. This assumption that programmers would prefer source code with lower surprisal underlines much of the work applying language models to code, but it has never actually been demonstrated via

controlled experiments. Our setting of comparing meaning-equivalent code expressions removes the possible influence of the algorithmic channel and allows us to design and execute two human subject experiments to discover if this assumption is accurate.

In each study, participants were asked to answer a series of forced-choice questions. We present them with two lines of source code, one which was the original developer-written code, and the other being a meaning-equivalent transformation of the expressions in that line of code. For example, one question asked:

> Please select which of the two following code segments you prefer:

```
outPacket = new byte[10 + length];
```

```
outPacket = new byte[length + 10];
```

Each question had identical phrasing, and we defined preference in our survey instructions as:

> By "prefer," we mean which snippet better reflects the way you'd prefer to code that expression, or reflects the form you'd find easier to read and understand.

In using this definition, we acknowledge the difficultly of disentangling the potential causes of production ease and audience design on the outcome of predictability.

Under this framework, we begin with the *n*-gram language models that were the primary focus of our corpus study. We wish to see if there is any relationship between surprisal and human preference at all, so we focus on examples where either the transformed or the original line is much more surprising to the model than the alternative. Then, we treat this decision as a binary without regard to the specific difference in surprisal—one of the two items in the pair is more surprising and one is less surprising. Thus, the research question underlying this study is:

> **RQ5.** *Do human preferences align with n-gram model judgments between meaning-equivalent lines of code with very different surprisal scores? How do these preferences vary over different types of transformations to code expressions?*

4.1. Materials

In both of our human subject studies, we limit the types of transformations and programming languages used from those that appeared in the corpus study when selecting our samples. First, we focus only on Java, which had both more data and a wider variety of transformations. Second, we select samples only from the relational/arithmetic swaps,

and parentheses adding/removing transformations. Variable name swapping is both much more difficult to test as it requires displaying entire functions to participants, and also the detrimental effects of altering variable names are already well established. Recall that we included them in the corpus study more as a validity check on the language models, as misnamed variables are known to be more surprising (to language models). In contrast, the other four transformations are not commonly established explicitly in style guides as previously mentioned in Section 3.3. Column (3) in Table 4 summarizes the transformations and language models used in this (and the following) human subject study.

Moreover, all examples used are samples from *real code*. They are not manufactured; each example consists of a line of original source code paired with a transformed version. However, not all expressions are suitable for presentation out of context to general programmers, so we applied filters when collecting suitable samples. We filtered out expressions containing hash computations as these were so repetitive that they reduced the diversity of samples (i.e., they all become related to hashes), or required more specialized knowledge (such as bit shift operations). We also removed lines that were too long (over 80 characters). In addition, the surprisal value we use is slightly different from the corpus study. When viewing the code, our participants will see the whole line (instead of just the changed expression) and see the line without any of the surrounding context. Therefore, we measure surprisal (a) using the entire line instead of just the tokens affected by the transformation and (b) without using the previous lines as context.

Our first study uses the *global n-gram model* from our corpus study and compares lines of code using average surprisal over shared tokens in the whole line instead of just the expression.[7] In this study, the goal was to evaluate if any agreement existed between language model judgment and human preference. Therefore, we sampled the transformations from the language model with the *most extreme* differences in surprisal—the top 20 single line transformations for each transformation type that most *increased* and *decreased* the surprisal compared to the original line. In this way, we look at the more common transformations that increased surprisal and the rarer ones that decreased it equally. After this initial selection (and the filtering previously described), we manually examined the samples and replaced a pair if it was too similar to another pair, when the pair had code that did not contain a complete expression (e.g., only part of a multiline expression), and cases where the transformation obviously disrupted the symmetry of the parentheses grouping in the line—for example, we would exclude transforming `(a == b) || (b == c) into a == b || (b == c) or (a == b) || b == c`. We would only keep cases where parentheses were added or removed around all the obvious groups. We avoided these cases as we believed that they are a particularly striking case of transformations that humans would overwhelmingly reject. By excluding them, we are looking at more challenging cases where preference might be harder to establish.

We choose the replacement samples from the top 40 most extreme cases, each time taking the largest difference that did not break with our described requirements. This gives a total of 160 transformation pairs evenly split across transformation types and on whether the transformation was considered harmful or helpful by the language model.

From these 160 pairs, we presented 80 to each user randomizing not only the order in which participants saw each item pair, but also the order of the pair in the multiple-choice selection—so sometimes the original version was the first option and sometimes the transformed version was.

4.2. Method

4.2.1. Procedure

We use Amazon's Mechanical Turk (*AMT*; Amazon Mechanical Turk & Inc., 2018) to select participants for our experiment. *AMT* has been used before to recruit subjects for programming studies (Alqaimi, Thongtanunam, & Treude, 2019; Chen, Stolee, & Menzies, 2019; Prana, Treude, Thung, Atapattu, & Lo, 2019). Since anyone can sign up with AMT, we selectively filter out a sample that can reasonably represent Java programmers. First, we follow recommended guidelines (Miele, 2018) for avoiding bots and poorly qualified workers; we require a 99% HIT acceptance rate, 1,000 or more completed HITs, and restrict workers to those in the United States and Canada. We also used Unique Turker (Ott, 2019) along with *AMT*'s own internal reporting to remove any repeat users. Next, we deploy a short qualification test that asks subjects to read some Java code and answer three comprehension questions; all three must be correctly answered to proceed to the main task. We tuned our comprehension questions with three pilot surveys. MTurkers were paid at a minimum wage rate ($12/h) for tasks they completed; the qualification test was estimated to take 5 min, and the main survey 20 min. Finally, at the end of the survey, we asked some demographics and open-ended questions about their strategies when selecting which segment they preferred.

In addition to the 80 questions we asked between pairs of source code lines, to measure subject attention, we included an unidentified attention check, which was a question like the others, except more obvious and incontrovertible. In the first survey, we asked if "`for(int i = 0; i < length; i++) {`" was preferable to "`for(int i = 0; length> i; i++) {`." We do not exclude those failing the attention check (it was only one question of many), instead using it as a measure for how attentive the subjects were overall. As long as failing the check is not common, we can be confident of reasonably attentive subjects.

4.2.2. Participants

Our survey had a total of 181 attempts across three batches, with 60 non-duplicate full completions of the survey. Of the 60, 50 passed the attention check, though there is little difference in overall agreement with the language model between those that passed and those that did not. Demographically, the mean age of our participants is 33.5 with *SD* 9.4. Our group had a mean of 6.1 and *SD* of 5.2 years of Java experience and mean 12.2 with *SD* 9.5 years of general programming experience, and they were primarily developers, students, and hobbyists who coded at least a few times a week. For gender and native speaker status, 50 of our participants self-identified as male, 8 as female, and 2 choose to

not answer, while 54 said they were native speakers of English and 4 said they were non-native. All but one had some college education, and most used AMT for extra income.

4.2.3. Modeling

To measure the relationship between human preference and language model preference, we first examined simple and majority vote agreement for each of the item pairs and Cohen's kappa (Cohen, 1960) for inter-rater agreement (using human majority vote as one rater and the language model as the other).[8] Then, we captured these effects in a more controlled fashion using a mixed-effects logistic regression. As the complexity of the random effects structure of our model caused the frequentist estimate to not converge, we estimate the model via Bayesian regression through the R package brms (Bürkner, 2017). Our presented model used the default priors of the package, but we validated convergence and alternative priors using the guidelines included in the WAMBS checklist (Depaoli & Van de Schoot, 2017) and followed general model validation advice for Bayesian statistics (McElreath, 2018). For each experiment, we first fit the full model design as justified by the experimental design, but we then checked for fit and compared against simpler models using the WAIC measure (Watanabe, 2010). When a simpler model is preferred by WAIC, we present results from the simpler model for ease of presentation and understanding. Our modeling and validation code is available via our replication package: https://doi.org/10.5281/zenodo.2573389.

4.2.4. Results

Overall, 61.9% (65.6% with majority vote, and 62.8% and 66.9% respectively for those passing the attention check) of the time our subjects agreed with the global *n*-gram model. The kappa value across all the items was 0.312 or *fair*. Fig. 3a groups the results by each item pair, broken down by transformation type. Humans overall agree with the language model for the swaps much more frequently than the parentheses transformations. Relational swaps have the highest agreement (indicated by values above 0.5). All of the disagreements are cases where the raters agreed with the original code (but the language model disagreed), suggesting a limitation in the language model rather than disagreement among coders. For parentheses, we also see a pattern: Our group tends to prefer variants with more parentheses (indicated by the reversed red/blue patterns in AddParen and RemoveParen; Fig. 3a), regardless of the language model preference. Moreover, the language model poorly predicts human majority vote preference for adding parentheses—agreeing only half the time.

Turning to the regression model, the formula is as follows: *Outcome* ~ *Surprisal* * *TransType* + (*1* + *Surprisal* * *TransType* | *Subject*) + (*1* |*Item*). The response variable *Outcome* is 1 if the human subject selected the original code, 0 for the transformation. The fixed effects are a binary predictor (Surprisal), which is 1 if the *language model* preferred the original line of code, and 0 otherwise, along with the type of transformation and their interaction term. We use the maximal random effects structure justified by the design (Barr, Levy, Scheepers, & Tily, 2013)—a random intercept by item pair, a random intercept and slopes for surprisal, transformation type, and their intercept

(a) Per Item Pair Majority Agreement Sorted by Rank

(b) Human Agreement vs N-gram Model Predictions

Fig. 3. Results of Experiment 1. (a) The fraction of agreement for each item pair by transformation, ordered from questions with the least agreement with the model to the most (40 per transformation). If more participants disagreed with the language model, the bar points down; if more participants agreed with the language model, the bar points up. Blue indicates the language model preferred the original way the code was written, whereas red indicates the transformed code had lower surprisal. The majority vote agreements using the *n*-gram surprisal by transformation (i.e., how many bars point up) are as follows: ArithmeticSwap (65%), RelationalSwap (80%), AddParen (50%), RemoveParen (67.5%). (b) A comparison of the model predictions with the human data by transformation type and whether the model prefers original versus transformed code. The *x*-axis shows when the language model preferred the transformed or the original version (the 0 and 1 codes for our binary predictor in the regression model).The *y*-axis shows the degree of human preference, with 0 being a tendency to prefer the transformed version, and 1 a tendency to prefer the original. The error bars are the model's prediction of this tendency, whereas each data point is the fraction of human preference for the transformed or original version *for each item pair*. This is equivalent to the majority agreement when the language model prefers the original and 1-majority agreement when the language model prefers the transformation.

by subjects. For priors, we use the defaults provided by *brms*. The transformation types are deviation coded, meaning the intercept value is the grand mean over all transformations. Thus, for the coefficients for the *parentheses removal* transformation, we subtract the three other type coefficients and provide it in the table for convenience. Finally, Bayesian estimates do not have *p*-values and confidence intervals in the same way as frequentist approaches. Instead, we report the equivalent 95% credible interval, the bounds of a probability distribution that has a 95% probability containing the regression coefficient. If this range is entirely above 0, it indicates a positive effect and vice versa for a range entirely below 0. To check robustness, we compared this model against a simpler model without the interaction effect, but our full model had a better WAIC score.

The coefficients of our Bayesian mixed-effects logistic regression are in Table 8, and Fig. 3b compares the model predictions against the actual data. As this is a logistic regression, the coefficients are log odds ratios. The odds ratio of the intercept by itself, $\exp(-0.60) = 0.55$, shows that when the language model prefers the transformation, our subjects are $1/0.55 = 1.8$ times more likely to also prefer the transformed line of code. By contrast, when the language model prefers the original code, our subjects are $\exp(-0.60 + 1.9) = 3.7$ times more likely to also prefer the original code. Importantly, the

crucial predictor of surprisal has a positive effect with 0 well outside its credible interval. Thus, on average, not only do humans agree with the language model more often than not, they agree with the language model almost twice *as strongly* in its judgments on the original code.

The transformation coefficients and interaction term in combination with the graphical depiction in Fig. 3b show very different behaviors among the different transformation types. As with the majority vote cases, the adding and removing parentheses show much weaker agreement that the two swapping transformations. If *n*-gram surprisal was perfectly aligned with human judgments, we would see values above 0.5 for the original and below 0.5 for the transformation on the logistic function. Instead, our subjects preferred more parentheses on everything, even when the code was not written that way. However, overall, human preferences align with language model assignments more often than not, giving some support for RQ5.

4.2.5. Discussion

More probable code, operationalized by surprisal from *n*-gram models, does demonstrate some alignment with programmer preferences, but this alignment is not strong and is not consistent across different transformations. For example, programmer preferences for less surprising parentheses variants are much less pronounced than they are among operand swaps. This effect could result from two possibilities. First, it could signal that human preferences for parenthesis are less pronounced—that they do not believe it impacts their understanding of these expressions. Alternatively, it could signal that the *n*-gram model assumptions of relevant context do not align well with what humans use for parentheses.

Therefore, acknowledging this potential confound, and having established that *some* effect exists at all using more extreme differences, we would like to see the impact of

Table 8
Fixed effects for Bayesian mixed-effects logistic regression using a binary difference in surprisal as a predictor and an *n*-gram language model

	Estimate	Error	l-95% CI	u-95% CI
Intercept	−0.60	0.16	−0.91	−0.28
Surprisal	1.90	0.25	1.41	2.37
AddParen	−2.03	0.59	−3.20	−0.91
Arithmetic	0.50	0.26	−0.01	1.00
Relational	0.16	0.26	−0.36	0.66
RMParen*	1.37	—	—	—
Surprisal:AddParen	−0.79	0.41	−1.63	0.01
Surprisal:Arithmetic	−1.02	0.37	−1.76	−0.29
Surprisal:Relational	1.64	0.44	0.81	2.53
Surprisal:RMParen*	0.17	—	—	—

*Parentheses removal (RMParen) does not get an independent coefficient estimate in deviation coding, so we calculate the implied coefficient.

using a stronger language model to demonstrate if a more nuanced relationship between surprisal and human preferences exists. Next, we will examine if surprisal from the transformer model, generally regarded as a superior to *n*-grams, both better aligns with human preference and better captures these preferences across different types of transformations. In the following experiment, we will see if preference aligns with a *gradient* of surprisal values instead of treating it just as a *binary* feature of lower and higher surprisal.[9]

5. Human study 2: Gradient transformer surprisal

5.1. Materials

We make three changes relative to the prior study. First, we use the *transformer model with BPE* that we used for validation in the corpus study as an example of a more powerful neural network language model. Second, we made a minor change to the surprisal metric. Based on the psycholinguistic literature demonstrating that reading times are proportional to surprisal (Smith & Levy, 2013), our surprisal predictor will now be the *total sum surprisal* of the line, rather than the surprisal of shared tokens (as in Section 4). This change only affects the parentheses transformations, as those are the only transformations in which we add and remove tokens. We compared this measure with the shared surprisal used in our prior experiment. The results were qualitatively similar,[10] but they are omitted from the full discussion; see the supplementary R Notebooks for details.

Finally, instead of drawing from the pairs with extreme differences in surprisal, we wished to see if the effect of preference was detectable along a *gradient* of surprisal values. We again selected positive and negative examples evenly from each of the transformation types, dividing these examples into buckets that span the range of surprisal values. We first selected a cutoff for the most extreme buckets for both increases and decreases in surprisal, such that these buckets would have 20 examples to draw from. Then, we divided the range between this cutoff and 0 uniformly, creating eight buckets that evenly divide the range of positive and negative surprisal changes. We initially randomly sample 2 from each bucket and replace symmetry breaking or overly similar examples with other randomly pulled examples from the same bucket. This gives 32 item pairs for each of the 4 transformations (128 total pairs), with surprisal values relatively uniformly distributed across all the range of possible surprisal values for a given transformation. The buckets are not used for analysis; they were only used to construct a balanced sample. As in the prior study, each participant saw a randomly selected half (64) of the items.

So, in summary, in this study we wished to answer the following research question:

RQ6. *Do language models recognized as more robust, such as the Transformer* (Vaswani et al., 2017), *show stronger agreement with human preferences? Moreover, if we consider a range of surprisal differences on a spectrum, will human subjects demonstrate greater agreement when the difference in surprisal is larger?*

5.2. Method

5.2.1. Procedure

The procedure for the second study was identical to the first with one exception. As we observed lower than expected agreement with our original attention check, we replaced it by instead asking if "`x = a + b + c;`" was preferable to "`x = (a) + (b) + (c);`." We believe this transformation is at least as unusual as our prior attention check, and thus participants should strongly prefer the first form.

5.2.2. Participants

The second preference survey included 211 attempts, with 59 successful non-duplicate completions of the qualification task. All 59 participants passed the attention check. The mean age of our participants in the second study was 32 with *SD* 10.1 years. For gender, 45 of our participants self-identified as male, 12 as female, 1 as non-binary/genderqueer/gender non-conforming, and 1 choose to not answer. For native speaker status, 55 said they were native English speakers of English and 4 said they were not. They had a mean of 5 and *SD* of 4.9 years of Java experience and mean of 9.6 and *SD* of 8.4 years of general programming experience, once again being mostly developers with the rest being students and hobbyists. The frequency of programming was similar to that in the previous study, with most coding at least a few times a week. Almost all of them coded at least a few times a week, with the remaining coding a few times a month. They mostly had at least a bachelor's degree, and use Mechanical Turk to supplement their income.

5.2.3. Modeling

We reuse the same modeling and validation procedures in this experiment as we did with the extreme *n*-gram surprisal experiment, which is described in Section 4.2.3.

5.2.4. Results

First, we consider the transformer model from the perspective of binary agreement, as with the prior experiment. On one hand, if the transformer is a better model of language than *n*-grams, as is widely believed, then it should better reflect human preferences. On the other hand, since we are sampling from throughout the distribution of changes in surprisal (rather than just the extremes), we would also expect greater uncertainty around the examples with smaller differences. Overall, with the total surprisal metric, MTurkers agreed with the transformer language model 66.60% of the time and 75% of the time with majority vote agreement, with a kappa value of 0.5 or *moderate*.

These majority vote agreements are broken down by transformation in Fig. 4a. The transformer is better than the *n*-gram model for all transformations other than the removing parentheses case. However, when comparing the two plots in Figs. 3a and 4a, we see that people tended to agree with the *n*-gram language models just when they preferred *more* parentheses. In comparison, the transformer judgments the subjects agreed with are far more balanced in preferences for the original and the transformed versions. Therefore,

it would seem that the transformer is a substantially better model than the *n*-gram model in terms of binary agreement, even when it is being penalized by drawing from all across the distribution of surprisal changes instead of just from the most extreme cases.

Next, we present our regression model to measure the effect of over a gradient of surprisal differences, shown in Table 9 and a fit versus data is shown in Fig. 4b.[11] This is another mixed-effects logistic Bayesian regression with a very similar formula to our prior experiment: *Outcome ~ LineDiff * TransType + (1 + LM_Out * TransType | ResponseId) + (1 | Item)*. The response variable is the same as before, 1 if the subject preferred the original line of code, 0 otherwise. In the fixed effects, we have replaced our binary representation of surprisal with *LineDiff*, the difference in total line surprisal between the transformed code and the original code. We retain the transformation type as the other fixed effect as well as the same random effect structure as before.

The main effect, *LineDiff*, has a positive effect with 0 outside its credible interval. This means that for an additional bit of surprisal, our MTurkers were 1.11 times more likely to prefer the original, confirming that for better language models like the transformer, incremental changes in surprisal can map to human preference. Though WAIC values indicate that including the interaction effect is better than excluding it, the differences and interactions of the transformation type all contain 0 in their credible intervals, indicating that the language model more consistently captured human preference across

(a) Per Item Pair Majority Agreement Sorted by Rank

(b) Human Agreement vs Gradient Model Predictions

Fig. 4. Results of Experiment 2. (a) The fraction of agreement for each item pair by transformation, ordered from questions with the least agreement with the model to the most. If more participants disagreed with the language model, the bar points down; if more participants agreed with the language model, the bar points up. Blue indicates the language model preferred the original way the code was written, whereas red indicates the transformed code had lower surprisal. The majority vote agreements using the transformer surprisal by transformation (i.e., how many bars point up) are: ArithmeticSwap (75%), RelationalSwap (81.3%), AddParen (78.1%), RemoveParen (65.6%). (b) A comparison of the model predictions with the human data by transformation type. The *x*-axis shows the range of differences in surprisal. The *y*-axis shows the degree of human preference, with 0 being a tendency to prefer the transformed version, and 1 a tendency to prefer the original. The lines and error are the model's prediction of this tendency, whereas each data point is the fraction of human preference for the transformed or original version *for each item pair*. This is equivalent to the majority agreement when the language model prefers the original and 1-agreement when the language model prefers the transformation.

Table 9
Fixed effects for Bayesian mixed-effects logistic regression using incremental differences in surprisal as a predictor and a transformer language model

	Estimate	Error	l-95% CI	u-95% CI
Intercept	0.44	0.12	0.20	0.67
LineDiff	0.10	0.01	0.08	0.13
AddParen	−0.44	0.33	−1.11	0.20
Arithmetic	−0.45	0.33	−1.10	0.21
Relational	0.13	0.20	−0.27	0.52
RMParen*	0.76	—	—	—
LineDiff:AddParen	−0.02	0.02	−0.05	0.02
LineDiff:Arithmetic	−0.01	0.02	−0.04	0.03
LineDiff:Relational	−0.01	0.01	−0.04	0.02
LineDiff:RMParen*	0 .04	—	—	—

*Parentheses removal (RMParen) does not get an independent coefficient estimate in deviation coding, so we calculate the implied coefficient.

different transformations. Looking at Fig. 4b, we can see that though the parentheses transformations tend to change surprisal less than the swaps (the range of points on the *x*-axis is smaller), the model generally approximates the patterns of human agreement versus the incremental surprisal differences well.

5.2.5. Discussion

We can confirm a positive answer for RQ6; surprisal as captured by a transformer model and human preferences demonstrate incremental degrees of agreement. The human preferences change as the surprisal changes; larger drops and increases in surprisal align with stronger human preferences. This helps resolve some of the questions raised by the prior experiment—whether the disagreement was a result of a lack of preference for predictable parentheses or due to misalignment between human preferences and *n*-gram model assumptions. With the transformer, we can see humans demonstrating consistent preferences for less surprising variants. There is still some variance in preferences between the transformations (though not significantly so according to our mixed-effects models); determining the true strength of preferences for predictable code expressions in different contexts will be an interesting future task (we expand upon this in Section 6.1). So, while *n*-gram surprisal has some correlation with human preference, transformer surprisal is a much better and more consistent metric of what code humans will find surprising. Carefully controlling for meaning, we see that surprisal *does* have validity as a measure of human preference for source code expressions.

6. General discussion

Our corpus and human subject studies show that programmers have preferences for more predictable code expressions, similar to preferences in predictability for natural

language. When controlling for meaning along the algorithmic (AL) channel, our corpus study shows that language model surprisal captures preferences for certain forms in the natural language (NL) channel. However, these preferences vary as some conventions are not as strong as others—particularly on when it is acceptable to remove nonessential parentheses. In our human subject studies, we see that on average the language model preferences align with explicit human preference judgments. However, that alignment is not strong in the weaker global *n*-gram, but the transformer is better able to consistently capture preferences across our range of transformations. Moreover, surprisal from the transformer can model the nuances of incremental surprisal difference and the degree of preference: The bigger the difference in surprisal, the more humans preferred the less surprising variant. This demonstrates not only that programmer preference corresponds to language model predictability, but a nuanced one where small differences in surprisal can matter. Our finding that developers prefer more predictable code forms supports the idea that the repetitiveness of code arises in part from human choices, and it demonstrates similarity in the processing of code and natural language.

6.1. Cognitive origins of preferences

What about the question of code's more predictable nature in comparison to natural language? The less repetitive nature of natural language may result from natural language very rarely being entirely meaning equivalent—since natural language has connotative as well as propositional meaning. For instance, while "bread and butter" is *prima facie* synonymous with "butter and bread," one might infer from a description of eating "butter and bread" that butter was unusually dominant in this situation. These subtle differences in meaning may drive speakers to choose different forms in different situations. However, `i = 1 + i` and `i = I + 1` are semantically equivalent on the AL channel. Perhaps because developers are trained to understand the true operational semantics code, different connotations for the above two seem unlikely to evolve. It is difficult to imagine how `i = 1 + i` might carry a subtly different and useful connotation, even on the NL channel, and thus it is difficult to find a situation (analogous to "butter and bread") where a developer might consciously feel the need to use that construction.

Another possibility is that repetitiveness is particularly beneficial in situations with increased cognitive load. Indeed, it has been proposed that children resort to regularization in language learning more so than adults specifically because they have reduced cognitive capacity (Hudson Kam & Newport, 2005; Schwab, Casey, & Goldberg, 2018). Because code comprehension is challenging, repetitiveness may be extra beneficial for code compared to natural language, such as the reduced cognitive efforts seen in repeated coding patterns. The first instance of these patterns has a spike of cognitive effort, and subsequent ones are much easier to process (Jbara & Feitelson, 2014, 2017). Cache language models also exploit this effect of local repetition; subsequent examples of patterns in the same file are easier to predict (Hellendoorn & Devanbu, 2017; Tu et al., 2014).

Where do these preferences originate from, and how do developers learn them? Research in natural language has demonstrated that preferences in natural language (e.g., "bread and butter" > "butter and bread") arise both from abstract/generative knowledge, which provides generalizable knowledge like "order a main dish before a condiment," and from direct experience with specific items or exemplars (Benor & Levy, 2006; Bybee, 2003; Goldberg, 2003; Hay & Bresnan, 2006; Morgan & Levy, 2016). These generative preferences may themselves be learned/generalized from exposure, or may arise from innate constraints on processing (e.g., a preference to put the more frequent word first may be motivated by availability in production; Bock, 1987; Ferreira & Dell, 2000). Future work could explore the extent to which programmers' preferences for certain forms of code are likewise driven by a combination of their innate processing preferences, learned generalized preferences, and direct experience with specific tropes (e.g., `i = I + 1`). Additionally, preferences may carry over from natural language. For example, a programmer might write `a + b` over `b + a` due to alphabetic ordering. Explicit instruction or style guides may also influence programmers' preferences, although (as described in Section 3.3), the specific transformations investigated in this paper are rarely if ever mentioned in explicit style guides.

Finally, while predictable code may be preferred by programmers overall, there may also be contexts in which predictability and preferences/comprehension genuinely do not align. Our results suggest that ordering preferences may be stronger than those for parentheses use. Programmers could choose to use more unpredictable code as a way to draw attention to a line or a segment, highlighting something important. For example, they could use a more descriptive but unusual iterator variable instead of "`i`" in an important loop. As language models improve and better capture what code is predictable, discovering these nuances for when predictable code is or is not beneficial and *why* will help structure cognitive models of programming.

6.2. Threats and generalizability

Establishing programmers' sense of and preference for predictability in code is challenging, and potentially can be conflated with various confounding factors. We address some potential threats and some factors of our design aimed to mitigate these threats.

Our combination of results from the corpora and human subject studies provides evidence of *consistency* in programmers' notions of predictability for code expressions. Our corpus study establishes that this notion of predictability is consistent between different projects: The probability distributions captured in one context carry over to another. These are largely consistent across different transformations, language models, and programming languages. Second, the combination of the corpus and the human subject study provides another layer of consistency between mature code that is written (corpus study) and that of the initial comprehension preferences of programmers outside these projects (human study). That we see consistency in preferences for more predictable code in these differing contexts suggests that this is not just an artifact of language models, but a real observable pattern in human cognitive preferences.

Though we cannot distinguish between predictable code expressions resulting from ease of production for the writer and code that is deliberately designed to be predictable for the reader's benefit, there is reason to seriously consider the audience-design–based account for code. Unlike natural language production, which is often studied in a spoken rather than written environment, code is both written *and* usually heavily edited and revised to work in a collaborative environment. Many projects adopt style guidelines to make submitted code easier to read, and open-source code often undergoes the process of *code review*, where team members may critique submissions for correctness or style (Bacchelli & Bird, 2013). Programmers are thus directly encouraged to write code with an audience in mind. Specific to language model surprisal, Hellendoorn et al. (2015) showed that when training language models on specific projects, new submissions that were more predictable relative to patterns learned on the project were more likely to be accepted by team members. This provides some evidence that language model surprisal can capture predictability as a measure of audience reception.

Moreover, as we mentioned, many of our transformations are not covered under explicit style guides and sometimes style guides can be conflicting. We were inspired to try the parentheses transforms based on the work of Gopstein et al. (2017, 2018) investigating code patterns that developers find confusing. They found that style guides advocating for minimal curly brace use—for example, when writing if statements with only one statement in the body, developers can choose to wrap the body in curly braces or not—sometimes conflicted with what developers found easiest to understand. This is consistent with our finding that parentheses are preferentially included to indicate evaluation order (even when not needed) to serve a similar role in segmenting expressions as curly braces do in control flow. Although the predictability of source code measured through surprisal and human understanding are different metrics, our results on parentheses removal suggest a similar phenomenon—that developers use them sometimes to benefit from easier readability.

Beyond the impacts of coding style, how generalizable are our results? We have chosen a reasonably diverse sample of projects in two widely used languages with a variety of language models, and our results largely hold up. However, we acknowledge that the comparison between Python and Java is limited to only one class of transformation—swaps around relational operators. We choose them because of their differences, though this limited how many transformations we would replicate. Replicating all the transformations in a language similar to Java or running a human study on Python may provide additional valuable insights.

While we believe it is likely that our results will generalize to languages similar to Java and Python, it is possible that other languages (e.g., Haskell) with tightly knit, highly skilled user bases may behave differently. Haskell is a purely functional language designed such that developers can write mathematically elegant and aesthetically pleasing programs, and real-world Haskell code is known to be less repetitive and predictable than other programming languages (Casalnuovo et al., 2019). This could lead to a community that places a greater focus on the expressivity of the language and implementing solutions artistically, similar to how programmers may experiment in "weird" languages (Mateas &

Montfort, 2005). Their motivations might be less focused on reducing cognitive load for themselves and for other readers and, as such, might open the possibility of more expressive alternative implementations. Therefore, as different programming languages are used in different contexts, the weights humans place on different factors while coding may influence the predictability of these language corpora.

We have also only focused on small-scale transformations to expressions. It is possible that larger transforms may have different effects and may require different modeling techniques. However, since we have demonstrated preferences even on these small expressions that are typically not described in explicit style guides, we expect larger transformations may demonstrate even larger effects. Nevertheless, our work gives a good grounding to future studies with larger transformations as it shows that even small changes to code can have an impact on surprisal and preference.

6.3. Actionability and future work

While our work thus far is more science than engineering, it does have practical implications. Though sound, meaning-preserving transforms are not realistic for natural language, they are for code! We confirm that the surprisal of code is associated with human preference, thus providing theoretical support for tools that aim to rewrite code into an equivalent form that might be preferred by programmers. Below, we highlight some applications in which surprisal can be leveraged and describe a roadmap for developing cognitive models of predictability in code.

6.3.1. Readability and complexity metrics

One area of much interest in which surprisal might be applied is that of readability. For natural language, obtaining readability metrics to classify the reading levels of texts has seen focus for over a century (DuBay, unpublished ms). For programming language, measures of software *complexity* have drawn on control flow, program length, vocabulary, and their combinations as a way to assess software (Halstead, 1977; McCabe, 1976) More recently, work on readability, as determined by human subjects beliefs about how difficult it is to understand code, has seen some attention (Buse & Weimer, 2008, 2009; Dorn, 2012; Posnett, Hindle, & Devanbu, 2011; Scalabrino, Linares-Vasquez, Poshyvanyk, & Oliveto, 2016). These models tend to be selected from hand-selected features; surprisal, as determined from good language models, may be able to act as a more accurate and scalable measure of readability.

However, determining how effective these models and metrics are for source code understanding can be challenging. Scalabrino et al. (2017) made a distinction between readability (the perceived ability of developers to understand code) and understandability (the actual ability to understand code) and found the two may not be as strongly aligned as hoped, though others have critiqued their modeling methods and found that combining existing metrics appropriately shows stronger correlations (Trockman et al., 2018). Kasto and Whalley (2013) compared these metrics against students' understanding of code, finding that they correlated with students' ability to correctly execute code, but not so with

their ability to explain the meaning of the code. Therefore, care must be taken in validating and applying surprisal in such contexts. This work provides a first step in this direction, but further work is needed to validate surprisal as a form of readability, both with different outcomes and contexts. For the former, we have recently published a study relating our meaning-equivalent expressions to comprehension accuracy and time, and we found that less surprising expressions take humans less time to compute (Casalnuovo, Morgan, & Devanbu, 2020).

6.3.2. Building tools and validating models

If surprisal can operate as a measure of readability, transformations that maintain meaning could create better code. In the long term, we envision surprisal as a metric to be used in automated style checkers, helping improve code on check in, or even used to generate code on a large scale. Likewise, this controlled environment provides another task in which language models can be compared against each other. Language models that aim to capture the structure of code, such as graph message networks (Allamanis, Brockschmidt, & Khademi, 2017), or more complex models that combine transformers with these graph models (Hellendoorn et al., 2020) may demonstrate further alignment with human judgment than the transformer by itself. Changing how context is processed or what context is used in language models and seeing how surprisal estimates align with human preferences and comprehension could help inform what combinations are most useful. Moreover, it would be useful to see how these judgments align with performance on various downstream software tasks.

6.3.3. Developing cognitive models of predictability

In the long term, our goal is to explore how the programmer's cognitive model of predictability is structured, and what implications this has for how programmers both read and write code. We analogize this goal to that of understanding natural language processing, in which the basic effect of predictability on comprehension has long been established, but where there remain many questions regarding how comprehenders determine what is predictable (i.e., the structure of the predictability model) and what pressures lead speakers to produce predictable utterances (e.g., speaker-centric pressures like availability vs. listener-centric pressures like audience design).

Similarly, we envision an extensive future research endeavor to understand comparable pressures in program comprehension and production. For example, in comprehension, one might explore different comprehension metrics such as explicit judgments (which we used here), comprehension speed and accuracy measures (Casalnuovo, Morgan, et al., 2020), eye tracking scanpaths, and neural measures. Stimuli might include naturalistic code fragments taken from corpus data for ecological validity, as we used here, or might include constructed examples designed to test specific hypothesized preferences (e.g., a preference for variables before literals, or preferences for parentheses in more vs. less complex expressions). Stimuli might explore local preferences within a single line of code (as we did here), or preferences for larger structure at the level of whole functions or whole files

(e.g., code databases like Rosetta Code [Mol, 2020] could provide examples of different algorithmic implementations to compare).

In production, one could similarly explore different types of stimuli as described above. One could also ask questions about whether these production preferences are due to speaker-internal pressures to produce what is easiest, or due to audience-design pressures to produce code that will be easy to read in the future. Noting that most uses of code are more like edited writing than like extemporaneous speech, one could also ask to what extent these production tendencies are produced during initial writing of code versus during later editing and code review.

In both comprehension and production, we can ask how these preferences develop as people learn to code, for example, which preferences emerge immediately (suggesting that they result from preexisting cognitive pressures, such as analogy to existing preferences in natural language, or innate pressures) and which only develop later as programmers become more experienced. For example, if preferences in code arise in part from frequency experience with specific forms (as in natural language), we would expect novice programmers to be less sensitive to these preferences. We note that the benefits of this kind of repeated exposure to specific forms are not exclusive to natural language, having been seen in other environments (such as chess; Chase & Simon, 1973). Regardless, we hope these results will be useful in designing better materials to teach programming, possibly through more explicitly showing what normative coding looks like.

We also believe these results will provide an interesting comparison case with natural language. As described in Section 1.2, code and natural language are subject to many of the same communicative pressures, but there are also important differences, including the existence of the algorithmic channel to constrain the meaning of code, the fact that code is more or less exclusively written (rather than spoken), and, relatedly, the fact that there are no native speakers of programming languages. A careful comparison of how preferences manifest in both code and natural language may elucidate which aspects of each system are driven by what they have in common versus the ways in which they differ.

Acknowledgments

This research is supported by NSF grant 1414172: "SHF: Large: Collaborative Research: Exploiting the Naturalness of Software." We thank the graduate and undergraduate students in the laboratory who helped test and provided feedback on early versions of our human subject studies.

Notes

1. The low cross-entropy that modern models provide over *unseen* corpora is evidence of their power.

2. An IDE is software that assists developers in programming and building applications, usually including a code editor, a debugger, and other tools. Popular examples of IDEs and programming languages they are commonly used with include Eclipse—usually used with Java, and Visual Studio—often used with C#.
3. Mutation testing is a technique that seeks to create semantically *different* programs to expose deficits in test suites (Papadakis et al., 2019), and though recent work has tried using surprisal to generate these transformations (Jimenez et al., 2018), they do not fit our equivalence requirements.
4. See https://bugs.eclipse.org/bugs/show_bug.cgi?id=405096.
5. Limitations in the Python transforms prevent an accurate count of the number of transformations, and so the first filter was not applied. We examined the comparable Java models without this filter as well but found little difference in the coefficients.
6. Surprisal difference plots for the LSTM and transformer models can be found in the supplementary R Notebooks.
7. For arithmetic and relational swaps, all tokens in the line are shared between the original and transformed code. For adding and removing parentheses, all tokens are shared except the set of parentheses in question. When designing the second human subject study, we made some adjustments to this metric; see Section 5.1 for details.
8. An alternative such as Fleiss's Kappa (Fleiss, 1971) for multiple raters does not work in our context as not every person saw every question.
9. We also performed an analysis using the transformer model surprisal scores on the samples used in this experiment. We omit these results from the text, but they are qualitatively similar and are available in our supplementary R Notebooks.
10. The shared and total metrics had an approximately similar agreement with humans, though the shared metric was better at predicting agreement on removing parentheses, and the total was better at predicting agreement on adding parentheses.
11. We also tried a regression model identical to that used in the *n*-gram experiment, where the transformer surprisal difference is treated as a binary lower/higher value. See the supplementary R notebooks for details.

References

Allamanis, M. (2019). The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 143–153). New York: Association for Computing Machinery. https://doi.org/10.1145/3359591.3359735

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, *51*(4), 1–37. https://doi.org/10.1145/3212695

Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to represent programs with graphs. *arXiv Preprint*, arXiv:1711.00740.

Allamanis, M., & Sutton, C. (2014). Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software EngineeringFSE 2014* (pp. 472–483). New York, NY: ACM. https://doi.org/10.1145/2635868.2635901

Alqaimi, A., Thongtanunam, P., & Treude, C. (2019). Automatically generating documentation for lambda expressions in java. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19* (pp. 310–320). Piscataway, NJ: IEEE Press. https://doi.org/10.1109/MSR.2019.00057

Amazon Mechanical Turk, Inc. (2018). Amazon Mechanical Turk. Available at: https://www.mturk.com/. Accessed April, 2019.

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 712–721). New York : IEEE. https://doi.org/10.1109/ICSE.2013.6606617

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, *36*, 81–95. https://doi.org/10.1145/163359.163375.

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. https://doi.org/10.1016/j.jml.2012.11.001

Benor, S., & Levy, R. (2006). The chicken or the egg? A probabilistic analysis of English binomials. *Language*, *82*, 233–278. https://doi.org/10.1353/lan.2006.0077

Bock, K. (1987). An effect of the accessibility of word forms on sentence structures. *Journal of Memory and Language*, *26*, 119–137.

Brandl, G., & Chajdas, M. (2020). Pygments syntax highlighter. Available at: https://pygments.org/. Accessed November 29, 2019.

Brooks, R. (1978). Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software engineering* (pp. 196–201). Atlanta: IEEE Press.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*, 543–554.

Broukhis, L., Cooper, S., & Curt Noll, L. (2019). The International Obfuscated C Code Contest. Available at: https://www.ioccc.org/. Accessed July 2019.

Bürkner, P.-C. (2017). brms: An R package for Bayesian multilevel models using Stan. *Journal of Statistical Software*, *80*, 1–28.

Buse, R. P., & Weimer, W. R. (2008). A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 121–130). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/1390630.1390647

Buse, R. P., & Weimer, W. R. (2009). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, *36*, 546–558. https://doi.org/10.1109/TSE.2009.70

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., Sharif, B., & 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). New York: IEEE. https://doi.org/10.1109/ICPC.2015.36

Bybee, J. (2003). *Phonology and language use* (Vol. 94). Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511612886

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)*. New York: Association for Computing Machinery. https://doi.org/10.1145/3377816.3381720

Casalnuovo, C., Morgan, E., & Devanbu, P. (2020). Does surprisal predict code comprehension difficulty? In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Toronto, Canada: Cognitive Science Society.

Casalnuovo, C., Sagae, K., & Devanbu, P. (2019). Studying the difference between natural and programming language corpora. *Empirical Software Engineering*, *24*, 1823–1868. https://doi.org/10.1007/s10664-018-9669-7

Castelhano, J., Duarte, I. C., Ferreira, C., Duraes, J., Madeira, H., & Castelo-Branco, M. (2019). The role of the insula in intuitive expert bug detection in computer code: an fmri study. *Brain Imaging and Behavior*, *13*, 623–637. https://doi.org/10.1007/s11682-018-9885-1

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, *4*, 55–81.

Chen, D., Stolee, K. T., & Menzies, T. (2019). Replication can improve prior results: A github study of pull request acceptance. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC '19* (pp. 179–190). Piscataway, NJ: IEEE Press. https://doi.org/10.1109/ICPC.2019.00037

Clark, H. H., & Murphy, G. L. (1982). *Audience design in meaning and reference, vol. 9 of Language and comprehension*. Amsterdam: North-Holland Pub Co. https://doi.org/10.1016/S0166-4115(09)60059-5

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, *20*, 37–46.

Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2003). *Applied multiple correlation/regression analysis for the behavioral sciences*. UK: Taylor & Francis.

Collberg, C., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand.

Collberg, C., Thomborson, C., & Low, D. (1998). Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 184–196). New York: ACM. https://doi.org/10.1145/268946.268962

Cook, R. D., & Weisberg, S. (1982). *Residuals and influence in regression*. New York: Chapman and Hall. https://doi.org/10.2307/2348833

Dash, S. K., Allamanis, M., & Barr, E. T. (2018). Refinym: Using names to refine types. In *Proceedings of the 26th ACM ESEC/FSE* (pp. 107–117). ACM. https://doi.org/10.1145/3236024.3236042

Demberg, V., & Keller, F. (2008). Data from eye-tracking corpora as evidence for theories of syntactic processing complexity. *Cognition*, *109*, 193–210.

Depaoli, S., & Van de Schoot, R. (2017). Improving transparency and replication in Bayesian statistics: The WAMBS-checklist. *Psychological Methods*, *22*, 240. https://doi.org/10.1037/met0000065

Dorn, J. (2012). A general software readability model. MCS thesis. Available at: http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf. Accessed April 27, 2018.

DuBay, W. H. (unpublished manuscript). The principles of readability.

Duraes, J., Madeira, H., Castelhano, J., Duarte, C., & Branco, M. C. (2016). Wap: Understanding the brain at software debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 87–92). IEEE. https://doi.org/10.1109/ISSRE.2016.53

Eclipse Foundation. (2020a). Eclipse java development tools. Available at: https://www.eclipse.org/jdt/. Accessed May 2018.

Eclipse Foundation. (2020b). Eclipse JDT language server. Available at: https://projects.eclipse.org/projects/eclipse.jdt.ls. Accessed May 2018.

Ehrlich, S. F., & Rayner, K. (1981). Contextual effects on word perception and eye movements during reading. *Journal of Memory and Language*, *20*, 641. https://doi.org/10.1016/S0022-5371(81)90220-6.

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. https://doi.org/10.1016/j.tics.2019.04.010

Ferreira, F., & Patson, N. D. (2007). The "good enough" approach to language comprehension. *Language and Linguistics Compass*, *1*, 71–83. https://doi.org/10.1111/j.1749-818X.2007.00007.x

Ferreira, V. S., & Dell, G. S. (2000). Effect of ambiguity and lexical availability on syntactic and lexical production. *Cognitive Psychology*, *40*, 296–340. https://doi.org/10.1006/cogp.1999.0730

Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological Bulletin*, *76*, 378. https://doi.org/10.1037/h0031619

Floyd, B., Santander, T., & Weimer, W. (2017). Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 175–186). IEEE. https://doi.org/10.1109/ICSE.2017.24

Gage, P. (1994). A new algorithm for data compression. *C Users Journal*, *12*, 23–38. https://doi.org/10.5555/177910.177914

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software*. Pearson Education India.

GitHub, Inc. (2020). GitHub. Available at: https://github.com/. Accessed May 10, 2018.

Goldberg, A. E. (2003). Constructions: A new theoretical approach to language. *Trends in Cognitive Sciences*, *7*, 219–224. https://doi.org/10.1016/S1364-6613(03)00080-9

Google and Gallup (2016). Trends in the state of computer science in US k–12 schools.

Google. (2020). Android API reference. Available at: https://developer.android.com/reference

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M.-K.-C., & Cappos, J. (2017). Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 129–139). ACM. https://doi.org/10.1145/3106237.3106264

Gopstein, D., Zhou, H. H., Frankl, P., & Cappos, J. (2018). Prevalence of confusing code in software projects: Atoms of confusion in the wild. In *MSR '18: 15th International Conference on Mining Software Repositories*, May 28–29, 2018, Gothenburg, Sweden. ACM. https://doi.org/10.1145/3196398.3196432

Hale, J. (2001). A probabilistic Earley parser as a psycholinguistic model. In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies* (pp. 1–8). USA: Association for Computational Linguistics. https://doi.org/10.3115/1073336.1073357

Halstead, M. H. (1977). *Elements of software science* (Vol. 7). New York: Elsevier.

Hay, J., & Bresnan, J. (2006). Spoken syntax: The phonetics of giving a hand in New Zealand English. *The Linguistic Review*, *23*, 321–349. https://doi.org/10.1515/TLR.2006.013

Hellendoorn, V. (2017). SLP-core. Available at: https://github.com/SLP-team/SLP-Core. Accessed May 2018.

Hellendoorn, V. J., & Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE* (pp. 763–773). https://doi.org/10.1145/3106237.3106290

Hellendoorn, V. J., Devanbu, P. T., & Bacchelli, A. (2015). Will they like this?: Evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15* (pp. 157–167). Piscataway, NJ: IEEE Press. https://doi.org/10.1109/MSR.2015.22

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., & Bieber, D. (2020). Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020*. Addis Ababa, Ethiopia: ICLR.

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, ICSE '12* (pp. 837–847). Piscataway, NJ: IEEE Press. https://doi.org/10.1109/ICSE.2012.6227135

Hollander, M., Wolfe, D. A., & Chicken, E. (2013). *Nonparametric statistical methods*, Malden, MA: (Vol. 751). John Wiley & Sons.

Hudson Kam, C. L., & Newport, E. L. (2005). Regularizing unpredictable variation: The roles of adult and child learners in language formation and change. *Language Learning and Development*, *1*, 151–195. https://doi.org/10.1080/15475441.2005.9684215

Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'reilly, U.-M., Bers, M. U., & Fedorenko, E. (2020). Comprehension of computer code relies primarily on domain-general executive resources. *BioRxiv*, 045732. https://doi.org/10.1101/2020.04.16.045732

Jbara, A., & Feitelson, D. G. (2014). On the effect of code regularity on comprehension. In *Proceedings of the 22nd international conference on program comprehension* (pp. 189–200). New York: Association for Computing Machinery. https://doi.org/10.1145/2597008.2597140.

Jbara, A., & Feitelson, D. G. (2017). How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering*, *22*, 1440–1477. https://doi.org/10.1007/s10664-016-9477-x

Jimenez, M., Checkam, T. T., Cordy, M., Papadakis, M., Kintis, M., Traon, Y. L., & Harman, M. (2018). Are mutants really natural? A study on how "naturalness" helps mutant selection. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 1–10). New York: ACM. https://doi.org/10.1145/3239235.3240500

Just, R. (2014). The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014* (pp. 433–436). New York, NY: ACM. https://doi.org/10.1145/2610384.2628053

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. https://doi.org/10.1145/3377811.3380342

Kasto, N., & Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. *Proceedings of the Fifteenth Australasian Computing Education Conference*, *136*, 59–65.

Kim, M., Bergman, L., Lau, T., & Notkin, D. (2004). An ethnographic study of copy and paste programming practices in oopl. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04* (pp. 83–92). IEEE. https://doi.org/10.1109/ISESE.2004.133489

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*, 97–111. https://doi.org/10.1093/comjnl/27.2.97

Kutas, M., & Hillyard, S. A. (1984). Brain potentials during reading reflect word expectancy and semantic association. *Nature*, *307*, 161–163. https://doi.org/10.1038/307161a0

Levy, R. (2008a). Expectation-based syntactic comprehension. *Cognition*, *106*, 1126–1177. https://doi.org/10.1016/j.cognition.2007.05.006

Levy, R. (2008b). A noisy-channel model of rational human sentence comprehension under uncertain input. In *Proceedings EMNLP* (pp. 234–243). Stroudsburg, PA: Association for Computational Linguistics. https://doi.org/10.3115/1613715.1613749

Levy, R., Bicknell, K., Slattery, T., & Rayner, K. (2009). Eye movement evidence that readers maintain and act on uncertainty about past linguistic input. *Proceedings of the National Academy of Sciences of the United States of America*, *106*, 21086–21090.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., & Poshyvanyk, D. (2014). Mining energy-greedy Api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 2–11). New York: Association for Computing Machinery. https://doi.org/10.1145/2597073.2597085.

Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M., & Sun, J. (2017). Stochastic optimization of program obfuscation. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on* (pp. 221–231). New York: IEEE. https://doi.org/10.1109/ICSE.2017.28

Madeyski, L., & Radyk, N. (2010). Judy–a mutation testing tool for Java. *IET Software*, *4*, 32–42. https://doi.org/10.1049/iet-sen.2008.0038

Mateas, M., & Montfort, N. (2005). A box, darkly: Obfuscation, weird languages, and code aesthetics. In *Proceedings of the 6th digital arts and culture conference* (pp. 144–153). Copenhagen, Denmark: IT University of Copenhagen.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

McElreath, R. (2018). *Statistical rethinking: A Bayesian course with examples in R and Stan*, New York: . Chapman and Hall/CRC.

Miele, J. (2018). *The bot problem on mturk*. Retrieved from http://turkrequesters.blogspot.com/2018/08/the-bot-problem-on-mturk.html

Mol, M. (2020). *Rosetta code*. Retrieved from http://www.rosettacode.org/wiki/Rosetta_Code

Morgan, E., & Levy, R. (2015). Modeling idiosyncratic preferences: How generative knowledge and expression frequency jointly determine language structure. In *Proceedings of the 37th Annual Meeting of the Cognitive Science Society*. Austin, TX: Cognitive Science Society.

Morgan, E., & Levy, R. (2016). Abstract knowledge versus direct experience in processing of binomial expressions. *Cognition*, *157*, 384–402. https://doi.org/10.1016/j.cognition.2016.09.011

Obaidellah, U., Al Haek, M., & Cheng, P.-C.-H. (2018). A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)*, *51*, 5. https://doi.org/10.1145/3145904

Ott, M. (2019). *Unique Turker*. Retrieved from https://uniqueturker.myleott.com/

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2019). Chapter 6 - Mutation Testing Advances: An Analysis and Survey. In A. M. Memon(Ed.), *Advances in computers*, , (Vol. 112, pp. 275–378). Amsterdam, The Netherlands: Elsevier. https://doi.org/10.1016/bs.adcom.2018.03.015

Peksag, B., & Maupin, P. (2019). *astor*. Retrieved from https://pypi.org/project/astor/

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*, 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

Posnett, D., Hindle, A., & Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories* (pp. 73–82). New York: Association for Computing Machinery. https://doi.org/10.1145/1985441.1985454.

Prana, G. A. A., Treude, C., Thung, F., Atapattu, T., & Lo, D. (2019). Categorizing the content of github readme files. *Empirical Software Engineering*, *24*, 1296–1327. https://doi.org/10.1007/s10664-018-9660-3

Python Software Foundation. (2020). *Python 3 Abstract syntax tree module*. Python Software Foundation. Retrieved from https://docs.python.org/3/library/ast.html

Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., & Hamze, A. (2010). Malware detection based on mining API calls. In *Proceedings of the 2010 ACM symposium on applied computing* (pp. 1020–1025). New York: Association for Computing Machinery. https://doi.org/10.1145/1774088.1774303.

Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., & Oliveto, R. (2017). Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 417–427). New York, NY: IEEE. https://doi.org/10.1109/ASE.2017.8115654

Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., & Oliveto, R. (2016). Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (pp. 1–10). New York: IEEE. https://doi.org/10.1109/ICPC.2016.7503707

Schmitt, N., & Carter, R. (2004). Formulaic sequences in action. In *Formulaic sequences: Acquisition, processing and use* (pp. 1–22). Amsterdam, The Netherlands: John Benjamins.

Schwab, J. F., Casey, L.-W., & Goldberg, A. E. (2018). When regularization gets it wrong: Children over-simplify language input only in production. *Journal of Child Language*, *45*, 1054–1072. https://doi.org/10.1017/S0305000918000041

Sennrich, R., Haddow, B., & Birch, A. (2016). 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. https://doi.org/10.18653/v1/P16-1162

Shaft, T. M., & Vessey, I. (1995). The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research*, *6*, 286–299. https://doi.org/10.1287/isre.6.3.286

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*, 219–238. https://doi.org/10.1007/BF00977789

Siegmund, J. (2016). Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 5, 13–20). New York: IEEE. https://doi.org/10.1109/SANER.2016.35

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., Saake, G., & 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). New York: ACM. https://doi.org/10.1145/2568225.2568252

Siegmund, J., Peitek, N., Parnin, C., Apel, S., Hofmeister, J., Kästner, C., Begel, A., Bethmann, A., & Brechmann, A. (2017). Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 140–150). New York: ACM. https://doi.org/10.1145/3106237.3106268

Smith, N. J., & Levy, R. (2013). The effect of word predictability on reading time is logarithmic. *Cognition*, *128*, 302–319. https://doi.org/10.1016/j.cognition.2013.02.013

Snedeker, J., & Trueswell, J. (2003). Using prosody to avoid ambiguity: Effects of speaker awareness and referential context. *Journal of Memory and Language*, *48*, 103–130. https://doi.org/10.1016/S0749-596X(02)00519-3

Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). Intellicode compose: Code generation using transformer. *arXiv Preprint*, arXiv:2005.08025

Tiarks, R. (2011). What maintenance programmers really do: An observational study. In *Workshop on software reengineering* (pp. 36–37). State College, PA: Citeseer.

Trockman, A., Cates, K., Mozina, M., Nguyen, T., Kästner, C., & Vasilescu, B. (2018). "Automatically assessing code understandability" reanalyzed: Combined metrics matter. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (pp. 314–318). New York: IEEE. https://doi.org/10.1145/3196398.3196441

Tu, Z., Su, Z., & Devanbu, P. (2014). On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014* (pp. 269–280). New York, NY: ACM. https://doi.org/10.1145/2635868.2635875

Vasilescu, B., Casalnuovo, C., & Devanbu, P. (2017). Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 683–693). ACM. https://doi.org/10.1145/3106237.3106289

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998–6008). Red Hook, NY: Curran Associates.

Vee, A. (2017). *Coding literacy: How computer programming is changing writing*. Cambridge, MA: MIT Press. https://doi.org/10.7551/mitpress/10655.001.0001

Von Mayrhauser, A., & Vans, A. M. (1993). From program comprehension to tool requirements for an industrial environment. In *[1993] IEEE Second Workshop on Program Comprehension* (pp. 78–86). New York: IEEE. https://doi.org/10.1109/WPC

Watanabe, S. (2010). Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research*, *11*, 3571–3594.

Weisstein, E. W. (2004). *Bonferroni correction*, From MathWorld--A Wolfram Web Resource. Available at: https://mathworld.wolfram.com/BonferroniCorrection.html. Accessed December 11, 2019.

Zhan, M., & Levy, R. P. (2018). Comparing theories of speaker choice using a model of classifier production in mandarin Chinese. *Association for Computational Linguistics*. https://doi.org/10.18653/v1/N18-1181

Zhong, H., Xie, T., Zhang, L., Pei, J., & Mei, H. (2009). MAPO: Mining and recommending API usage patterns. In *European conference on object-oriented programming* (pp. 318–343). Berlin: Springer. https://doi.org/10.1007/978-3-642-03013-0_15