Is “Naturalness" a result of deliberate choice?

Kevin Lee
Department of Computer Science, UC Davis
Davis, CA
kllee@ucdavis.edu

Casey Casalnuovo
Department of Computer Science, UC Davis
Davis, CA
ccasal@ucdavis.edu

ABSTRACT

Programs operate on dual channels: not only are they meant to be run via a compiler and interpreter; they are also meant to be read and understood by humans. However, when people try and understand the meaning of a program, it's unlikely that they are doing so by imitating a compiler or interpreter, *viz.,* mentally simulating operational semantics. More likely, they are probabilistically guessing the meaning, using some sort of mental model reflecting how they expect meanings to be *typically* implemented as programs. For this to work quickly and efficiently, program authors must be deliberately restricting their choices on how meanings are typically implemented - so that given a program, code readers can quickly guess at its meaning. In other words, the conditional distribution of implementation given meaning must be rather skewed. To test this theory, we run an experiment applying several *semantics preserving transformations* (operator swapping and parenthesis addition/removal) to Java expressions to see if developers prefer a more limited set of forms for conveying the same meaning. We find that these transformations generally produce “unnatural" programs, supporting the theory of a skewed conditional distribution of implementation given meaning.

CCS CONCEPTS

• **Theory of computation** → *Program semantics*; • **Computing methodologies** → *Lexical semantics*;

KEYWORDS

Dual channel model, Language modeling, Code transformation

ACM Reference Format:
Kevin Lee and Casey Casalnuovo. 2018. Is “Naturalness" a result of deliberate choice?. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND BACKGROUND

The flexibility of programming languages give developers a potentially limitless pool of resources and methods in order to solve problems across numerous application domains. However, despite this flexibility, research has uncovered that in practice, the language of source code tends to repetition, greater repetition in fact than ordinary natural human language. This observation of the repetition in human written software, which Hindle *et al* [7] call *naturalness*, has provided great gains in applications, since language modeling techniques historically developed for *Natural Language Processing (NLP)* often work even better in the source code domain. They have been used to improve code completion [6, 11], complete APIs [8], fix source code [9], recover variable names [10, 12], and many more applications.

However, while these models often port successfully to the code context, this by no means indicates that their naive adoption is the best or even a good way of applying them to code. As Allamanis *et al.* point out in their survey of the field, one of the greatest challenges faced in this area is determining what adaptions to these models are necessary to best improve code[1]. In particular, one way in which source code has additional complexity over natural language is the two purposes it serves. Traditionally, source code is thought as being interpreted and run by the machine, but as code must be developed and maintained by teams, the code also serves as a form of communication *between humans*. This notion, which Allamanis *et al.* describe as the *dual channel model*[1], poses an interesting issue in relating code's meaning and its implementation. In the machine channel, two programs that are semantically equivalent can be run identically, swapping one version out for another. Humans can determine meaning by executing the program as a computer might, but we can also conceive of the human channel of communication as based on a probabilistic relation between meaning and implementation.

Why might humans approach code in this probabilistic manner? Source code is known to be more repetitive than natural language, and some preliminary work suggests that at least some of this repetition comes from human choices [3]. We theorize that humans find reading and writing code inherently difficult, and thus use repeating, familiar patterns to help make code easier to read. Specifically, this would imply that *given a particular computation C, programmers will favor one implementation over others.* Stated more formally, if $I_1, I_2, ..., I_n$ are different, equivalent implementations of the same underlying computation C, then

$$\exists j : p(I_j|C) \gg p(I_i|C) \forall i = 1, ..., n, i \neq j \qquad (1)$$

If this equation is true, the distribution of implementation given meaning is highly skewed, and it would be much easier for experienced developers to draw on their experience and reuse existing solutions. For example, this logic of reusing existing solutions is often described as basis of design patterns commonly employed in

[1]Though similar notions about code's dual nature have existed in research for a long time[2].

software. Thus, we wish to try and describe this relationship in a quantifiable manner.

To capture this relationship between implementation and meaning, we describe an experiment to take human written Java programs and use *semantics preserving transformations* to change code expressions into meaning equivalent, but differently implemented forms. If, as the theory expects, the distribution of programs given meaning is very skewed, then we would expect language models trained on a large corpus of Java to strongly prefer the original developer written source code.

Running this experiment on a small set of expression-level transformations - swapping numerical operators for *+, ** and adding and removing parentheses, we have found that this theory generally holds - but that there are some exceptions that offer the opportunity to reformat code to be more predictable.

2 METHODOLOGY

2.1 Data and Language Modeling

In this experiment, we use 12 Java projects as the training data. These projects were selected by choosing popular starred projects on GitHub and then manually selecting projects from diverse domains. In addition to the GitHub projects, we also include part of the Eclipse core source code. For our case studies, we choose 3 projects for testing - Biojava, the Apache Commons Math Library, and the Spring Framework from Pivotal. Since our transformations focus heavily on expressions and especially numerical expressions, we biased our selection to find projects with many examples of such mathematical expressions for our test set.

We use 4 variants of language model to capture the probability of a sequence of code tokens. For our language models, we use a basic 6-gram model with Jelinek-Mercer smoothing that we denote as the 'global' model. To measure the local patterns in code, we also use an ngram-cache model, as originally described by Tu *et al.* [11]. Then, to consider the effects of the transformations on underlying source code patterns, rather than the effects of specific variable name choices, we consider an alternate training and testing corpus where we use the Pygments[2] syntax highlighter to replace all identifiers and types with generic token types, and literals with a simplified type[3]. All our models are implemented in the SLP-Core framework, by Hellendoorn *et al.* [6][4]. We measure the predictability of our expressions through the standard measure of entropy, comparing the average entropy of tokens that appear *only in both the original and the transformed version of the expression.* This average uses only the tokens involved in the changed expression - not the whole line.

We record for modeling purposes the starting line number of the transformation, average depth of the transformed nodes from the root of the Abstract Syntax Tree (AST), the number of tokens in the line, and the number of transformations performed in the line. The type or operator of the transformed expression's parent node is also recorded, along with a summary of operators used in the expression (the most and least common - see 2.2).

2.2 Semantic Transformations

We present three kinds of semantically equivalent transformations in this paper: 1) swapping of the operands in mathematical infix expressions, 2) adding parentheses, and 3) removing parentheses. While training the models, each Java source file from the training projects is parsed using the AST Parser from the Eclipse Java development tools (JDT) and the visitor pattern is used to count the type of each node. The operator type is counted for nodes that are infix, prefix, and postfix expressions. These counts are used to identify the most and least commonly used node types or operators that appear within a transformed expression. After training the Java language models, we iterate over all Java files in the test projects, excluding unit test files by ignoring filepaths that contain the string "test".

For each file from the test projects, the AST is used to perform the possible transformations. We transform only one line at a time, resetting the AST to its original, untransformed state before the next transformation is made. The visitor pattern is used to identify infix expression nodes for possible transformations.

For the swap transformation, only infix expressions with the plus (+) and times (*) operators involving operands of double, float, int and long data types are visited for possible transformations. Infix expressions with more than two operands are only transformed if the data type of the operands are int or long to avoid accuracy errors due to floating point precision limitations. Infix expressions with 2 operands are simply swapped. The number of random permutations considered for infix expressions with more than 2 operands is equal to the number of operands in the infix expression.

The parentheses adding transformation relies on the tree structure of the AST to add parentheses while preserving the correctness in the order of operations. For each transformation we randomly select a subset of infix subexpressions to add parentheses. We cap the maximum amount of possible transformations by the number of infix expression sub-nodes in the tree, including the original one in the count. Parentheses are not added to expressions whose parent is a parenthesized expression to avoid creating double parentheses. Parentheses are also never added around the entire expression. Any duplicate transformations are ignored, but still count towards the maximum number of permutations considered.

For the parentheses removal, the visitor pattern is used to identify parenthesized expressions that are sub-nodes of infix expressions. Each parenthesized expression is then passed to the Necessary-ParenthesesChecker from the Eclipse JDT Language Server to check if the parentheses are needed. This is the same algorithm used by the "Clean Up" feature within the Eclipse IDE. In the removal case, we generate all possible removal permutations, but limit our selection to lines with no more than 4 potential locations to avoid exponential growth in the number of transformations.

3 RESULTS

While trends appear in our data, the details of the interactions between language model and transformation are somewhat complex. Given space limitations, we will provide a brief summary of our high level results thus far combined with a few deeper case studies. In our surface level examination, we will use a combination of box plots and paired t-tests with Cohen's D effect sizes to compare

[2]http://pygments.org/
[3]For example, we keep 1,2,3 and replace numbers with labels like <int> and <float>.
[4]https://github.com/SLP-team/SLP-Core

the average entropy values of expressions in the original and in the transformed code. As the distributions of average expression entropy are approximately normal, the t-test is an appropriate measure of comparison. In one of the case studies, we will describe some observations from our regression models; we omit the full model details for space. In total, we had 8,608 swap transformations across 7,444 lines, 20,874 parenthesis adding transformations across 9,541 lines, and 2,670 parenthesis removing transformations across 1,356 lines.

Table 1: Directed effect sizes of paired t-tests comparing the entropy of the original code with the transformed code. A negative effect size indicates that the transformation *decreases* entropy. All tests were significant with p < .001.

	Global	Cache	Global Type	Cache Type
Swapping	0.313	0.859	0.360	0.706
Add Paren	-0.858	0.142	0.366	0.869
Remove Paren	1.034	0.940	0.087	0.361

Figure 1 gives an overview of the effect of the transformations with each of our language models: a generic ngram model, an ngram model with a cache effect, and these models trained and tested on corpora with abstracted types and literals. On the far left, we see the results of swapping operands for both addition and multiplication, the center - adding parenthesis to expressions, and the far right - removing parenthesis. The effect sizes associated with the differences between the original and transformed entropy are located in Table 1.

Consider first the case of swapping the operands for multiplication and addition, represented by Figure 1a. Using a global model, there is an entropy increase associated with a *small* effect size. However, when using the cache model, the effect of this increase is much greater with a *large* effect size. This increase is expected, as the cache model weights local patterns more highly. Seeing no difference in this model would indicate that the developers were using patterns in the same file inconsistently (such as alternating $a + b$ and $b + a$) - a very unusual occurrence. The difference between the global and cache model reoccurs in the models with abstracted variable and literal types - though in both cases the effect is smaller. This reduction is likely the effect of more predictable patterns overall in both the abstracted corpora; variable names tend to be the least predictable tokens.

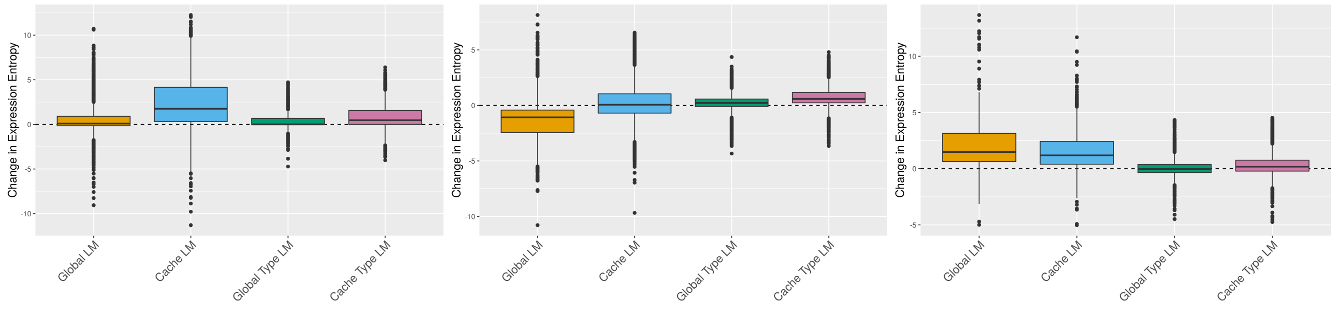
Providing a more detailed look, Figure 2 displays the relationship of the average entropy of the original code versus the change in entropy caused by the transformation. We have used a basic global ngram model with the original variable names included. Here we see a general downward trend - in cases where the original entropy of the line is low, transforming tends to make the line less predictable. But when the line is highly entropic already, transforming the line can in fact reduce the entropy. This downward pattern also appears (even more strongly) in the adding parenthesis case, but not in the removing parenthesis case. This suggests that some transformations may be helpful for more surprising and less common tasks, though preserving existing parentheses is preferable. In such cases, developers might have fewer pre-existing solutions and thus try more novel patterns.

The case of adding parenthesis provides perhaps the most complex and interesting story, as Figure 1b suggests. Using only a global model, we see that adding parenthesis to developer written code actually *reduces* the expression entropy. Not only that, but Table 1 shows that this is a *large* effect size. When accounting for the local patterns with the cache model, this reversal goes away. In the abstracted variable name models, there is an increase of entropy after transformation, once again with the cache model having a larger effect as expected. So how can we interpret this mixture of results? For one, the effect of the choice of variable names is significant. Since adding new parenthesis tends to increase entropy when considering only the structural patterns, we conclude that these projects are using variable names seen in the training corpus that use *different* parenthetical patterns.

We speculate that this suggests that given some specific structural pattern, for example perhaps two expressions joined by a logical OR, developer choices about whether to wrap the constituent components are not consistent given particular variable names. One developer may choose to write *(a > 0) || (b >0)* whereas another may write *a > 0 || b > 0* without a strong preference between them. However, in general the form *(VAR > VAR) || (VAR > VAR)* may be preferable to the version without parenthesis.

While we are still exploring the actual patterns these preferences fall along, we present a brief summary of a linear regression model of global regression model's entropy change. The explanatory variables used are the expression's original entropy, the number of transformations, the average AST depth of the transformations, the number of tokens in the line, the line number in the file, the operator of the parent node to the transformations, and the most common operator in the expression (as described in the methodology). We filter the response variable for outliers using the $1.5 * IQR$ metric, check model diagnostics, and check that normalized variance inflation scores are less than 5. Though we omit details for space, we note that that model has R^2: 0.711. All coefficients except the line number were significant; we list variables that explain at least 1% of variance, a (+) indicating an increase in the variable corresponds with an increase to the response variable and (-) indicates the opposite. In decreasing order of variance explained, these variables are: 1) the original entropy of the expression (-), 2) the number of transformations (+), 3) the number of tokens in the line (+), and 4) the type of the parent node. While we are interested understanding how specific operators in the expression effect the transformation, their effect size is small, so we avoid any specific interpretation. However, we are still exploring the effects and interactions of operator type with different language models and plan to expand our analysis as we gather more data.

Now consider the case of removing parenthesis in Figure 1c. Unlike the previous examples, the effect of the transformation using a cache model is actually slightly smaller than the global model - though both effects are still considered *large.* When abstracting identifiers, the size of these effects are dampened to *neglible* and *small* and more in line with prior cache observations. At the most basic level, one possible interpretation here is that when developers choose to include unnecessary parentheses for good reason. The abstracted models suggest that the particular variable names may be a more important factor in this decision than the underlying structural patterns themselves. In the regular models, the cache



(a) Swapping (b) Adding Parenthesis (c) Removing Parenthesis

Figure 1: Comparisons of the average entropy difference between the transformed source code and the original expression. Positive values indicate the transformation increased the entropy and negative values indicate the transformation reduced entropy.

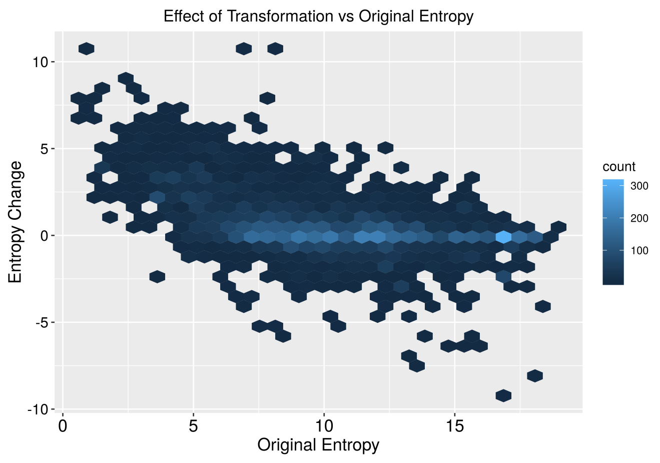


Figure 2: Hexbin plot comparing the original entropy of an expression prior to transformation by swapping numerical +/∗ operands and the difference in entropy between the original and transformed expression. Higher counts indicate higher point density. The model is an ordinary 6 gram model with no cache and no identifier or literal abstraction.

model detecting a smaller difference than the global model suggests that the usage of vocabulary together with parentheses tends to be very consistent across different files and projects. It seems that with parenthetical expressions, previously observed effects of locality may not hold as strongly[11].

Finally, we note the inspiration for the parentheses transformations came from related work on programming understanding and confusing code[4, 5]. Gopstein *et al.* found conflicts between style guides advocating for using only necessary curly braces and actual understanding of code. Parentheses serve a similar role in segmenting expressions as curly braces do in control flow. Though the predictability of source code and human understanding are different metrics, we believe our results on parentheses may hint at a similar conflict, that developers use them inconsistently and possibly could benefit from their increased usage.

4 DISCUSSION

Our initial foray into understanding the distribution of human choices in source code suggest that the conditional distribution of implementation given meaning *is* skewed. However, there are cases where this does not hold - and this provides opportunities to reformat code into more normalized forms. We plan to expand this work to 1) look at additional and larger transformations such as shuffling lines, 2) compare the model evaluations of code predictability with human studies of readability and understandability, and 3) develop tools to rewrite code to be more understandable.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182* (2017).

[2] Ted J Biggerstaff, Bharat G Mitbander, and Dallas E Webster. 1994. Program understanding and the concept assignment problem. *Commun. ACM* 37, 5 (1994), 72–82.

[3] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2018. Studying the Difference Between Natural and Programming Language Corpora. *arXiv preprint arXiv:1806.02437* (2018).

[4] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K-C Yeh, and Justin Cappos. 2017. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 129–139.

[5] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 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, 11 pages. https://doi.org/10.1145/3196398.3196432

[6] Vincent J Hellendoorn and Premkumar Devanbu. 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.* 763–773.

[7] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

[8] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15).* IEEE Press, Piscataway, NJ, USA, 858–868. http://dl.acm.org/citation.cfm?id=2818754.2818858

[9] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* ACM, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[10] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM,

New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[11] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/2635868.2635875

[12] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 683–693.