Code Phonology

An exploration into the vocalization of code

Hermans, Felienne; Swidan, Alaaeddin; Aivaloglou, Efthimia

DOI
10.1145/3196321.3196355

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the 26th Conference on Program Comprehension, ICPC 2018

Code Phonology: An Exploration into the Vocalization of Code

Felienne Hermans, Alaaeddin Swidan
Delft University of Technology
f.f.j.hermans,alaaeddin.swidan@tudelft.nl

Efthimia Aivaloglou
Open University of the Netherlands
fai@ou.nl

ABSTRACT

When children learn to read, they almost invariably start with *oral reading*: reading the words and sentences out loud. Experiments have shown that when novices read text aloud, their comprehension is better then when reading in silence. This is attributed to the fact that reading aloud focuses the child's attention to the text. We hypothesize that reading code aloud could support program comprehension in a similar way, encouraging novice programmers to pay attention to details. To this end we explore how novices read code, and we found that novice programmers vocalize code in different ways, sometimes changing vocalization within a code snippet. We thus believe that in order to teach novices to read code aloud, an agreed upon way of reading code is needed. As such, this paper proposes studying *code phonology*, ultimately leading to a shared understanding about how code should be read aloud, such that this can be practiced. In addition to being valuable as an educational and diagnostic tool for novices, we believe that pair programmers could also benefit from standardized communication about code, and that it could support improved tools for visually and physically disabled programmers.

CCS CONCEPTS

• **Social and professional topics → Computing education**; **K-12 education**;

ACM Reference Format:
Felienne Hermans, Alaaeddin Swidan and Efthimia Aivaloglou. 2018. Code Phonology: An Exploration into the Vocalization of Code. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196355

1 INTRODUCTION

Everyone that has ever seen a young child that has just learned to read, knows they do not read like adults. Young children read aloud, not just to demonstrate their newly acquired skill, but also because they simply cannot do it in a different fashion yet. Most children take years to learn to read silently, during which they go through a number of phases including whispering and lip movement. Several studies have shown that, for novice readers, reading aloud supports comprehension [10]. This should not come as a surprise, even expert readers on their native language sometimes fall back to this behavior when reading text that is difficult, or written in a foreign language they have not fully mastered yet.

We observe that, in learning how to program, no attention is given to the pronunciation, or rather the *phonology* of code, i.e. the way we read code aloud.

When teaching programming, we simply assume that students can read the code silently, in their head, while also processing what the code does, an activity known as *tracing*. We do not practice the vocalization as a skill in isolation, leading to various different ways in which code is pronounced by programmers. The fact that vocalization is not standardized might, inadvertently, impose a high cognitive load on novice learners, since they have to both read the code in their head, and 'execute' the functionality of the code, an activity often referred to as 'tracing' [11]. We propose exploring the idea of *code phonology*, ultimately leading to consensus in the field about how code snippets should be pronounced.

This can be challenging, even for simple statements. For example, how should we pronounce an assignment statement like `x = 5`? Is it *"x is 5"*? Or *"set x to 5"*? Or *"x gets 5"*? And what about an equality check? Is it *"if x is is 5"*? Or *"if x is 5"*? Or *"is x is equal to 5"*?

We see two distinct phases in this process. Firstly, language communities will have to define a shared phonology for their languages, answering the vocalization question for simple statements like the ones above, but also for higher level concepts like classes and modules. Discussing questions of the vocalization like the ones above can reveal how we think about code. In our exploratory experiments, we found that teenage novices show little consensus in how to read code, so there is a lot to learn from. For example, one participant read a function application as *"f of x"* while reading the definition of that function as *"f takes x"* which might lead to a great discussion in language design whether they should or should not be the same character. After this phase in which vocalization is explored, we envision that communities would converge on a shared, unambiguous phonology of their language. While this may seem like a daunting task, we think it compares to how programming language communities agreed on style guides, even after languages had been in use.

We hypothesize that such a shared phonology of source code will be useful in teaching in a variety of ways. Firstly, it helps teachers to read code in a systematic way, allowing them to explain more precisely, and removing the need for conversations like *"You need a bracket there. No a round one""*. Secondly, reading code aloud could be a diagnostic tool, helping teachers to assess what learners have understood from their reading of a piece of source code. Finally, we expect the activity of reading code aloud to contribute to better understanding of what the code does, following findings from natural language acquisition.

In addition to uses in education, we see broader applications too. An agreed upon understanding of how code is vocalized can help

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196355

pair programmers in their communication, as well as visually and psychically impaired developers in hearing and dictating code to their programming interface.

2 BACKGROUND AND MOTIVATION

Our main motivation for this work is observing the struggles novices have in learning to program. When teaching or pairing, we have observed programmers learning a new language spending effort on communicating clearly. *"A bracket. Now a curly one. And then a closing bracket. Yes a round one."* Such statements can be heard in programming classes worldwide. We seem to be spending insufficient effort on how to vocalize code. In that, the field of programming clearly differs from the field of natural language acquisition, in which reading aloud or *oral reading* as it is more commonly referred to in literature, is studied extensively. For example, researchers have found that comprehension of text is better when reading aloud [10]. According to literature, this is due to the fact that, when reading aloud, children are less likely to skip parts of the text, leading to better understanding [5].

This made us wonder if reading aloud would also improve program comprehension, but for that, we need a shared way of pronouncing code, for which we coin the term *code phonology*. The focus of this paper is to explore how we would define and use such a phonology. In future full length papers we plan to run studies to measure the effect of vocalizing code on comprehension.

3 A FIRST EXPLORATORY EXPERIMENT

To explore the idea of code phonology, we conducted a first exploratory experiment with novice programmers reading Python out load. The subjects were 10 Dutch high school students: 8 boys and 2 girls participated in the experiment, all between 11 and 13, with an average of 12. At the time of the experiment, the students had received about 20 hours of Python lessons. We asked them to read the some snippets of Python code aloud [1] in such a way that a student that knows Python would be able to type the code, a method similar to the one used by Begel and Graham [2].

3.1 Observations

Reading the code in a consistent fashion proved a daunting task to our 10 participants. All of them read at least one symbol, keyword or variable in an inconsistent way. Children aged 11 to 13 should not have any issues in reading their native language in a consistent way, and even with English they should be able to read with relative ease. The fact that they struggle and make conflicting decisions strengthens our belief that an agreed upon way to read code could help them comprehend it. In the following subsections we will zoom in on some of these inconsistencies, but the fact that they appear in itself is interesting.

Natural language effects In addition to generic inconsistencies, we saw specific ones. For example, participants struggled with choosing between Dutch and English reading. None of the children in the study were native speakers of English. Most of the children (8) had Dutch as their native language, 2 were bilingual children naming Turkish as their first language, however still being fluent in Dutch.

[1]Read the program here: https://pastebin.com/n36Upp1p

While reading the code, some children used Dutch pronunciation of English words, saying *"wheel"* rather than *"while"*. Two children spelled out `if` in Dutch, saying it as *"ie-ef"*. The most interesting natural language effect though was on the variable `i`. In Dutch, `i` is read as *"ee"* as in "to breed". In English of course it is read as *"ai"* like in "fry". Just one child pronounced the variable in a consistent way (the Dutch way). The other 9 all mixed the Dutch and English vocalization, sometimes even within the same code block.

A different effect was that on word order. One of the subjects changed the order of the words to form a proper Dutch sentence. Rather than saying *"if temperatuur is 20"*, he said *"als temperatuur 20 is"* which is grammatically correct.

Symbols Since these students did not learn about lists yet, we did not include statements with lists and symbols [] in the reading exercise. The symbols included were: (,), ==, !=, <, += and =. An interesting vocalization (which we hardly ever encountered with professional developers) is to say == as *"is is"*, which 7 children in the experiment did. There were also other ways that the symbols confused the children. The symbol < was vocalized in many different ways, including smaller than and lower than, but also arrow or bracket, or was skipped entirely.

One participant struggled specifically with combinations of two symbols like != and ==, and systematically only reading the first symbol. This behavior is likely to be influenced by their experience in mathematics where there usually is only one symbol, and that carries over as an assumption into programming. Misconceptions often move from one field to another in such a way [3].

Syntactic versus semantic level Some children read things that were not technically in the code, adding meaning. One participant read `for i in range(0,15)` as *"for i in a range 0 comma 15"*, adding the meaning that the numbers occur in *a* range and making the sentence more like a sentence in natural language. Another participant read `def kwadraat(x):` as *"def function kwadraat x colon"* adding the meaning that this is a function definition. That might be good practice for novice learners, or a hint to language designers that function is a better keyword.

Omissions There were symbols that were omitted by students consistently. Some did not read the double prime symbol (*""*) anywhere, which is somewhat reasonable since it occurred only in print statements and is thus somewhat implied. One student omitted *":"* consistently which can also be said to be implied after an `if` or a `while`. Most students, however, were inconsistent and omitted some symbols selectively. One of the most common omissions are the open brackets and the colon in the first snippet, which comprises a for loop ranging from 0 to 15. It was was read as *""for i in range 0 comma 15""* by 5 participants, who however all did pronounce the brackets or the colon in other snippets.

3.2 Oral Reading and Programming Capabilities

Even though the scale of this experiment is small, we observed that the students that were able to read the programs more consistently and more 'semantically' were also the students that were ranked as the best programmers by their teacher. Of course, an open question here is whether the students read better because they comprehend better or the other way around.

3.3 Summary

In the exploratory study we have seen confusion over how to pronounce source code, confirming our hypothesis that how to read code is not a given.

We observed students struggling with reading keywords, pronouncing variables and symbols. Even though we did not explicitly measure cognitive load in our experiment, it seems that students were spending energy on deciphering symbols, such as < and ==, on what to read and what to skip, and—in case of bilingual learners—on choosing between English and their native language. Cognitive load theory [17] suggests that to free up mental room for more complex thoughts, easier processes must be *automated*. For example, before reading full words at once, children first automate the skill of reading letters. Before being able to process large multiplications, children need to have automated additions. We believe that the cognitive load spent on deciding how to read a variable or keyword, cannot be spent on comprehension, and must thus be automated by separate practice, making room for the more complicated process of metal execution of a program, often called 'tracing' [11].

4 CODE PHONOLOGY

Based on research into learning natural language, and on the above described experiment, we believe that a focus on reading code aloud is needed. As such, our goal is for programming language communities to converge on a common vocalization of their programming language, which could have the form of a mapping from syntax elements to their sounds. We use the word *vocalization* here, rather than *verbalization* as previous papers have used, since we can imagine that communities would use sounds rather than words to express syntax elements. Why not use a tongue click for a curly bracket?

We envision two steps in this process: establishing the phonology, and using it. In the following subsections we sketch these phases. We, however, are looking for feedback of the program comprehension community on how to shape these phases.

4.1 Establishing the phonology

Before we can use the phonology for teaching, it needs to be standardized. While our end goal is to put the phonology to use, we do believe that its *creation* has value itself. For example, studying where syntax and sound disagree, gives us insight into how people interpret code, and how they give meaning to syntax. As described in Section 3, some participants vocalized the open bracket differently dependent on its context. A participant would say *"f of x"* when reading a function call, while that same participant used *"f takes x"* when reading the definition of a function. This gives rise to the question of whether a function call and its definition should be represented with the same symbol. On one hand, using the same symbol seems logical because of the close relationship between definition and application. On the other hand, if many programmers vocalize it differently, is it really a good choice?

Similarly, programming languages can have symbols written in different ways, but vocalized the same by many developers. For example, both = and == are read in the same way by some experienced developers. Some picked *"is"* while others said *"equals"*, but they used the same term for all these, and never used *"is-is"* as novices did. That gives rise to the question of whether these should be different symbols. A study by Stefik and Siebert [16] that showed that it is best for programmers to use = for both assignment and equality checks seems to confirm the way programmers pronounce symbols. That gives credibility to our hypothesis that pronunciation can give insights into the quality of programming syntax.

We are however aware that the question of how to pronounce code can be seen as controversial, since one can argue that code is meant to be written by humans and executed by machines. But can we really comprehend and communicate source code if we cannot vocalise it? When exploring the shared phonology, we expect to gain a deeper understanding on program comprehension from a fresh angle.

While reaching a shared phonology seems very far away, we imagine the process similar to that of agreeing on a style guide, with rules for naming and whitespace.

4.2 Using the phonology

After the establishment of the shared phonology, we see several uses for it.

Firstly, we envision a use in programming education, for both learners and teachers. For teachers, the phonology could be a way to communicate more efficiently with learners. Agreeing on how to vocalize keywords and symbols eases communication. We also see reading code aloud as a valuable tool for diagnosing learners' understanding of code. For reading natural language, this is seen as an important aspect of reading aloud; it transforms what is being read into an observable artifact [6, 9, 13]. As a result, it becomes simpler for educators and researchers to reflect upon the student's understanding of the text. We hypothesize that this will be a benefit of reading code aloud too. It can be hard for novice programmer to clearly articulate what they have understood about a programming concept and its execution, hampering a meaningful exchange with teachers. Reading code aloud might ease this process. In some cases, the vocalization of code could help educators identify *misconceptions* novices hold about programming. A misconception is an incorrect understanding of a concept, leading to mistakes in writing or reading programs [15]. For example, imagine *"x becomes 5"* being the agreed upon reading for the assignment of 5 to x, and a student reads it like *"x is 5"*. This might mean that the student focuses on the symbol of the = as known from mathematics class, and the associated meaning of equality, rather than on storing a number.

We also see value in using the activity of reading code. We expect it to engage children in programming a fresh way, connecting programming more to reading natural language than to mathematics, a potentially more inclusive frame. With a shared and agreed upon phonology, learners can practice reading the code separately from comprehension, much like learners of a language first practice speaking and reading the "a" and only then use it in words, which could result in lower cognitive load when tracing programs [11, 17].

In addition to novices, we see broader usage too. A group of people that could benefit from phonology are adult professional programmers engaging in (distributed) pair programming. Research shows that defective communication is one of the four causes of the pair dismissal [4] and that vocal dialog helps the pair cooperate

more realistically than with written means. Pairs often also need audio communication in order to search for information and solutions on the web and use them for modifying the code [14], and therefore pair programmers could benefit from a consistent way to 'speak' code to each other. A code phonology could also be useful when designing tools for programming, since an agreed upon method to read code aloud would allow computers to also read the code consistently, possibly aiding blind and visually impaired programmers. Inversely, there are programmers that can read but not easily write code because they lack full control over their arms. A phonology of code will ease the dictation, since it defines how a language should be pronounced. IDE's or programming-by-voice tools (e.g. [2, 7, 12]) could offer more powerful dictation support by taking advantage of that.

5 RELATED WORK

We are not the first to explore how code could sound. Related to our current research question are two lines of work. The most related are programming-by-voice tools. Inspired by the need to provide an alternative input method for programmers suffering from repetitive stress injuries, after observing that *"spoken programs contain lexical, syntactic and semantic ambiguities that do not appear in written programs"*, Begel and Graham designed Spoken Java, a semantically identical variant of Java that is easier to say out loud [2]. Other natural language interfaces for programming include NaturalJava [12] and VoiceCode [7]. While some of these papers, most notably [1] also describe experiments in which developers read code aloud, the goal of these papers was to create a version of Java that could be spoken and do not further explore the issues in vocalization and its effect on comprehension. For example, an issue like the context dependent vocalization of (was not in the scope of these papers.

Another category of work related to ours is work on program auralization, an idea firstly coined by DiGiano and Baecker [8]. Program auralization is the idea to use non-speech audio to increase the ease by with programmers comprehend source code, for example by playing a note for every execution of a loop to quickly hear how often is was executed, or to lower the note in case of recursive calls to a function. Some experiments [18, 19] have demonstrated that program auralization can help novice programmers to comprehend code, indicating that 'hearing' code, albeit different from how we propose, can be an aide for code comprehension.

6 CONCLUSION AND OUTLOOK

In this paper we propose to start working towards a standard *phonology* of programming languages, which prescribes how to read source code aloud. We performed an exploratory study with 10 novices reading code, in which we observed subjects struggling with reading the code, and a preliminary link between code reading and performance in programming.

The difficulties we observed among these non-native English learners of code have increased the confidence in our idea that in teaching attention should be devoted to how to read source code aloud. The fact that better programmers also read code in a more consistent way indicates that reading code aloud could be a useful diagnostic instrument, and potentially even increase code comprehension.

We have a broad group of future studies in mind. Firstly, we want to measure the correlation between program comprehension and consistency of oral reading. Furthermore we plan to measure the *quality* of oral reading in a more systematic way to be able to compare that to comprehension too. Subsequently, we plan to design a teaching method for oral reading of code and measure its effect on program comprehension in a controlled study. In parallel, we will explore various different phonologies, their attractiveness to programmers and the ease with which they can be taught to both novice programmers and experts.

REFERENCES

[1] A. Begel and S. L. Graham. 2005. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. 99–106. https://doi.org/10.1109/VLHCC.2005.58

[2] A. Begel and S. L. Graham. 2006. An Assessment of a Speech-Based Programming Environment. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 116–120. https://doi.org/10.1109/VLHCC.2006.9

[3] B. Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[4] G. Canfora, A. Cimitile, and C. A. Visaggio. 2003. Lessons learned about distributed pair programming: what are the knowledge needs to address?. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* 314–319. https://doi.org/10.1109/ENABL.2003.1231429

[5] Peter F. de Jong and David L. Share. 2007. Orthographic Learning During Oral and Silent Reading. *Scientific Studies of Reading* 11, 1 (2007), 55–71. https://doi.org/10.1080/10888430709336634

[6] Ryan Deschambault. 2011. Thinking-Aloud as Talking-in-Interaction: Reinterpreting How L2 Lexical Inferencing Gets Done. *Language Learning* 62, 1 (2011), 266–301. https://doi.org/10.1111/j.1467-9922.2011.00653.x

[7] Alain Désilets, David C. Fox, and Stuart Norton. 2006. VoiceCode: An Innovative Speech Interface for Programming-by-voice. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (CHI EA '06)*. ACM, New York, NY, USA, 239–242. https://doi.org/10.1145/1125451.1125502

[8] Christopher J DiGiano and Ronald M Baecker. 1992. Program auralization: sound enhancements to the programming environment. In *Proceedings of the conference on Graphics interface'92*. Morgan Kaufmann Publishers Inc., 44–52.

[9] Andrea D. Hale, Renee O. Hawkins, Wesley Sheeley, Jennifer R. Reynolds, Shonna Jenkins, Ara J. Schmitt, and Daniel A. Martin. 2010. An investigation of silent versus aloud reading comprehension of elementary students using Maze assessment procedures. *Psychology in the Schools* 48, 1 (2010), 4–13. https://doi.org/10.1002/pits.20543

[10] Sherry Kragler. 1995. THE TRANSITION FROM ORAL TO SILENT READING. *Reading Psychology* 16, 4 (1995), 395–408. https://doi.org/10.1080/0270271950160402

[11] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[12] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces (IUI '00)*. ACM, New York, NY, USA, 207–211. https://doi.org/10.1145/325737.325845

[13] Suzanne M Prior and Katherine A Welling. 2001. " Read in Your Head": A Vygotskian Analysis of the Transition from Oral to Silent Reading. *Reading Psychology* 22, 1 (2001), 1–15.

[14] Till Schummer and Stephan Lukosch. 2009. Understanding Tools and Practices for Distributed Pair Programming. 15, 16 (oct 2009), 3101–3125.

[15] J. Sorva. 2012. Visual program simulation in introductory programming education. PhD Thesis, Aalto University. (2012), 428 pages.

[16] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages. https://doi.org/10.1145/2534973

[17] John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction* 4, 4 (1994), 295 – 312. https://doi.org/10.1016/0959-4752(94)90003-5

[18] Paul Vickers and James L Alty. 1996. Caitlin: A musical program auralisation tool to assist novice programmers with debugging. ICAD.

[19] Paul Vickers and James L. Alty. 2005. Musical Program Auralization: Empirical Studies. *ACM Trans. Appl. Percept.* 2, 4 (Oct. 2005), 477–489. https://doi.org/10.1145/1101530.1101546