How do Scratch Users Name Variables and Functions?

Author1
Uni1
Address1
Email: email1.com

Author2
Uni2
Address2
Email: email2.com

***Abstract*—The abstract goes here.**

I. Introduction

The naming of identifiers in software code is considered, by several researchers, as one important aspect of its quality. Previous studies have shown a correlation between the proper naming of identifiers and the readability and comprehension of the code. Comprehension is linked to maintainability tasks and efficient performance of developers while performing them. As one scientist once said *"the code should be considered for human and less occasionally for a computer to execute"*.

In computer science and programming education, there seems a focus on the programming concepts and the syntax of the languages. Programming courses give less attention to naming variables and identifiers. One of the most common examples in many programming languages is the use of *"foo"* and *"bar"* naming for variables and functions. These two identifiers have meaningless names, and to some extent, they represent a refusal to name, which give the learner the conception that the name is less important, or irrelevant, to the programming task.

Recent studies have focused on the analysis of naming in software code repositories. In this paper, we give attention to novice programmers with the Scratch visual language. When it comes to novice programmer and learners, it is more important for the software community to understand conceptions in naming that are prevalent. These users will be the future developers, and at some point, they need to follow an implicit or explicit guideline of naming and collaborating in larger software repositories. To this end, we have analyzed 240,000 Scratch projects. Scratch is a block-based visual language that was developed by MIT with the aim of helping young people learn the basic concepts of programming and collaboration. Scratch has recently become very popular among school-age children and introduced as part of the curriculum in some countries as a means to teach programming. Analyzing Scratch programs will give researchers and software engineering community a perspective on naming from learners making their first step in programming and might turn to be the future programmers. In this paper we aim at answering the following research questions:

RQ1 How do Scratch users name variables and procedures?
(a) The length distribution of variable and function identifiers across projects.
(b) The usage of single-letter variables.
RQ2 How do Scratch users use language-dependent features in naming identifiers? In particular, to what extent are the following usage patterns popular within Scratch projects?
(a) Using spaces within identifiers
(b) Using numeric digits as identifiers
(c) Using the parameter in the middle name for procedures]

More questions: analysis of the peak value of 5, what is the most recurrent name? Is it expected or justified? More questions: variable length based on category of the Scratch program: game animation, ... More questions: variable length in relation to Dr. Scratch CT

II. Background and motivation

Naming identifiers in software code have been studied extensively in the past decades. in practice, identifiers are a major part of software code as one study found that within the Eclipse code, which has 2 MLoC, 33% of the tokens and 72% of characters are dedicated to identifiers. For a human to read that code, it is crucial to understand what the identifier means, and then can deduce what the code does. With no surprise, several studies have confirmed the link between good identifier naming and code's readability and comprehension. The comprehension is not a target in its own; the true reason is that better comprehension lets the developers perform maintenance tasks more effectively and efficiently. But what is a good naming approach? It is out of this paper's scope to explore the various recommendations of good identifier naming. However, it is worth to mention that there are different perspectives on what a good name is. Some researchers emphasize the usage of actual and complete words from a dictionary, or known abbreviations, which reflect the context of the program's purpose. Others argue that consistency in naming style is the most important. The usage of single-letter identifiers attracts much attention in research. For programmers, it is tempting to choose single-letter identifiers for quicker code writing that involves less mental load on the choice of a name. Additionally, single-letter named variables such as *"i"* or *"j"* have become almost a standard choice for index values when coding loops. Research has shown however that shorter identifier names are longer to comprehend, and the length of a variable should reflect its scope.

Despite the agreement on the importance of identifier names, and efforts to introduce naming guidelines and additional tools to help the programmer choose better names, developers find giving appropriate names to identifiers as a difficult task. In the end, it is the decision of the individual writing the code, and many factors may contribute to that decision. One area to consider here is the computer science and in particular programming education. There seems to be a great focus on the programming concepts, while less to zero attention is given to beautifying the code. For example, the usage of the identifier names *"foo"* and *"bar"* is prevalent in code examples. These names have no real meaning, and the student cannot link them to what the code does.

We are inspired by the work done by **ref** where they explored the single-letter naming for multiple software repositories from different programming languages. It is important for the software community to understand the patterns in which software developers apply naming to identifiers. We argue that this understanding can improve the quality of the guidelines and introduce research-supported tools that meet the needs of the programmers. For this sake, we believe it is even more important to focus on the novice programmers, students, and learners. For these future developers, the usage of a particular naming pattern in this level forms a (mis)conception that could move along with them to other programming languages and future careers. The work was done by **add ref** excludes the visual languages which have recently become a favorable choice for elementary and high schools as an introduction to programming, for example, Scratch and Alice. These block-based languages allow the user to create and assign their identifiers for variables and blocks. By studying these identifiers, we can understand better how novice programmers apply standard and language-specific naming, and how it compares to other textual based languages previously explored in the literature.

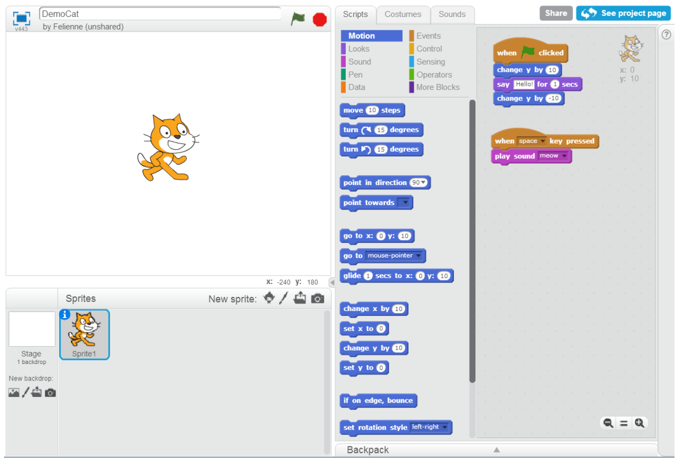
III. Relevant Scratch Concepts

This paper is by no means an introduction into Scratch programming, we refer the reader to [1] for an extensive overview. To make this paper self-contained, however, we explain a number of relevant concepts in this section.

Scratch is a block-based programming language aimed at children, developed by MIT. Scratch can be used to create games and interactive animations, and is available both as a stand-alone application and as a web application. Figure 1 shows the Scratch user interface in the Chrome browser.

A. Sprites

Scratch code is organized by 'sprites': two-dimensional pictures each having their own associated code. Scratch allows users to bring their sprites to life in various ways, for example



```
when [green flag] clicked
change y by 10
say Hello! for 1 secs
change y by -10

when space key pressed
play sound meow
```

Fig. 1. The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'motion' in the middle and the code associated with the sprite on the right. The upper right corner shows the actual location of the sprite.

by moving them in the plane, having them say or think words or sentences via text balloons, but also by having them make sounds, grow, shrink and switch costumes. The Scratch program in Figure 1[1] consists of one sprite, the cat, which is Scratch's default sprite and logo. The code in the sprite will cause the cat to jump up, say "hello", and come back down, when the green flag is clicked, and to make the 'meow' sound when the space bar is pressed.

B. Events

Scratch is *event-driven*: all motions, sounds and changes in the looks of sprites are initiated by events. The canonical event is the 'when Green Flag clicked', activated by clicking the green flag at the top of the user interface. In addition to the green flag, there are a number of other events possible, including key presses, mouse clicks and input from a computer's microphone or webcam. In the Scratch code in Figure 1 there are two events: 'when Green Flag clicked' and 'when space key pressed'

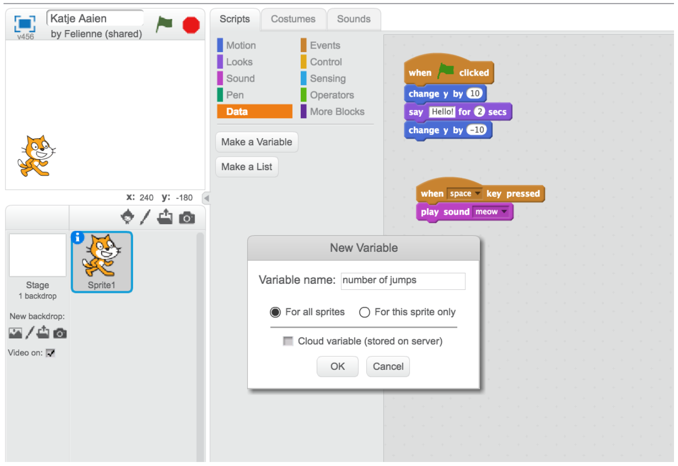
C. Scripts

Source code within sprites is organized in scripts: a script always starts with an event, followed by a number of blocks. The Scratch code in Figure 1 has two distinct scripts, one started by clicking on the green flag and one by pressing the space bar. It is possible for a single sprite to have multiple scripts initiated by the same event. In that case, all scripts will be executed simultaneously.

D. Remixing

Scratch programs can be shared by their creators in the global Scratch repository[2]. Shared Scratch programs can be 'remixed' by other Scratch users, which means that a copy

[1]https://scratch.mit.edu/projects/97086781/
[2]https://scratch.mit.edu/explore/projects/all/



```
when [flag] clicked
change y by 10
say Hello! for 2 secs
change y by -10

when space key pressed
play sound meow
```

New Variable

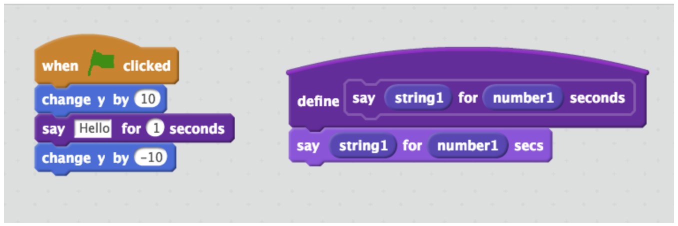
Variable name: number of jumps

[✓ For all sprites] For this sprite only

Cloud variable (stored on server)

OK Cancel

Fig. 2. The Scratch user interface to create a variable



```
when [flag] clicked
change y by 10
say Hello for 1 seconds
change y by -10

define say (string1) for (number1) seconds
say (string1) for (number1) secs
```

Fig. 3. Scratch code to define and invoke a procedure

of this program is placed in the user's own project collection, and can be then further changed. The 'remix tree' of projects is public, so users can track which users remix their programs, a bit similar forking in GitHub. Contrary to forking however, changes upstream cannot be integrated back into the original project. Because Scratch users can remix each others programs, maintainability of programs is important. A given Scratch program can be read and adapted by many children, there are projects on the Scratch home page which are remixed hundreds of times. **maybe this part about importance should be move somewhere else? this section might be skipped.**

E. Variables

Like most textual languages, Scratch users can use variables. Variables are untyped, but have to be 'declared' through the Scratch user interface, shown in Figure 2. This figure also shows that, contrary to most programming languages, variable names in Scratch may contain spaces.

F. Procedures

Scratch also allows users to create their own blocks, called procedures. They can have input parameters, and labels in between them. Functions are created with an interface similar to the one to create variables. Figure 3 shows the definition and invocation of a procedure.

IV. Research design and dataset

A. Overall design

here we need to tell something about replications, see the paper by Shull et al on importance of replications in software engineering.

B. Dataset

C. Data analysis

To augment the visual comparison of the variable name data derived from Scratch as well as the programming languages considered by Beniamini et al. [2], we conduct statistical analysis.

Understanding differences in variable name lengths requires comparison of multiple distributions, traditionally performed as a two-step process consisting of (1) testing a global null hypothesis, that can be intuitively formulated as "all distributions are the same", using ANOVA or its non-parametric counterpart, the Kruskal-Wallis test, and (2) performing multiple pairwise comparisons of different distributions, testing specific subhypotheses such as "distributions 2 and 4 are the same". However, it has been observed that such a two-step approach can result in inconsistencies when either the global null hypothesis is rejected but none of the pairwise subhypotheses is rejected or vice versa [3]. Moreover, it has been suggested that the Wilcoxon-Mann-Whitney test, commonly used for subhypothesis testing, is not robust to unequal population variances, especially in the unequal sample size case [4] Therefore, one-step approaches have been sought. We opt for one such approach, the $\widetilde{\mathbf{T}}$-procedure of Konietschke et al. [5]. This procedure is robust against unequal population variances, respects transitivity, and has been successfully applied in empirical software engineering [6], [7], [8]. In particular, we use the Tukey (all-pairs) contrasts to compare all distributions pairwise.

Furthermore, to understand differences and similarities between the distributions of single-letter variable names in different languages we represent each programming language as a 26-dimensional vector with the dimensions corresponding to 'a', ..., 'z', and apply hierarchical clustering. **why hierarchical?**

V. Results

This section presents an overview of our analysis of variable and function name use in our previously published Scratch dataset **ref ICER** .

Unless indicated otherwise, all graphs show the number of projects a certain type of variable is used in, rather then the number of occurrences of the variable. Some projects access and update variables hundreds of times, so we felt that counting use would skew the graphs towards bigger programs. **maybe this paragraph fits with dataset more than with results? Not sure.**

Figure 4 shows the distribution of lengths in the corpus. **still need to add the 20+ bucket if we want to have that**

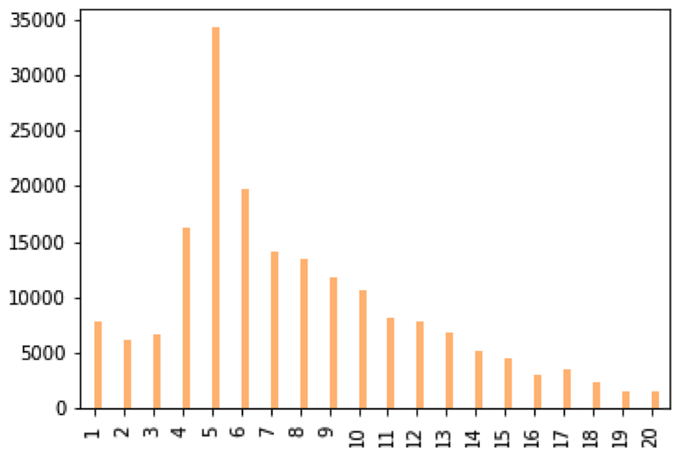


Fig. 4. Number of projects using variables of different lengths

A. One letters

Figure 5 shows the distribution of variables of one letter in the corpus.

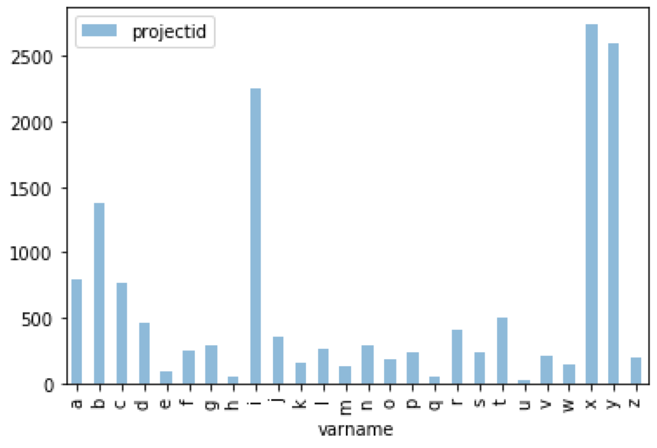


Fig. 5. Number of projects using variables of each of the one letter variables

Figure 6 shows the distribution of *functions* of one letter in the corpus.

The ICPC paper **add proper reference and cite** also explored the types of one letter variables, by performing a questionnaire among **summary of study here** .

This gave us the idea to also explore types of one letter variables in the context of our dataset. While Scratch variables have no types, we can deduce their use from our dataset, by exploring what assignments are made to variables and deducing the types from those. For example, the two variables in Figure 7 represent a string and an integer respectively. With this process we can compare our results to the types of **ICPC paper** . Figure 8 shows the distribution of variables of one letter in the corpus.

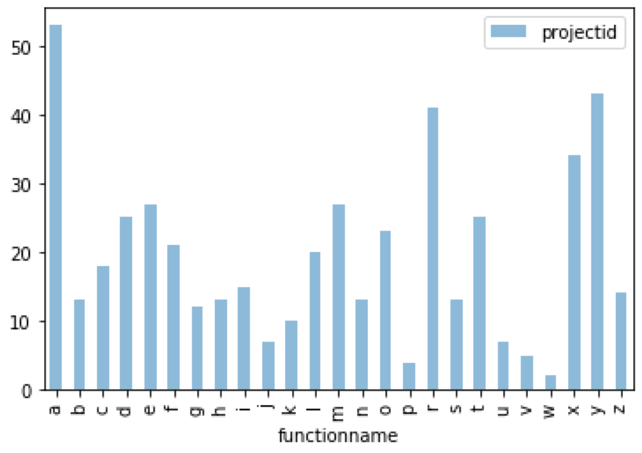


Fig. 6. Total occurrence of functions of one letter

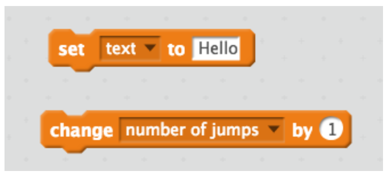


Fig. 7. Two variables, one of type string and one of type integer

here we need to write some reflection on the results. Alaaeddin....? :-) Comparing the results with the other languages and between functions and variables.

VI. Scratch specific contructs

A. Use of spaces in variable names

Contrary to most textual programming languages, Scratch allows users to use spaces in variable names. This is quite commonly used, about 30.000 projects use one or more variables with a space in it, versus 60.000 that use only space-free variable names. Figure 9 shows the distribution of spaces in

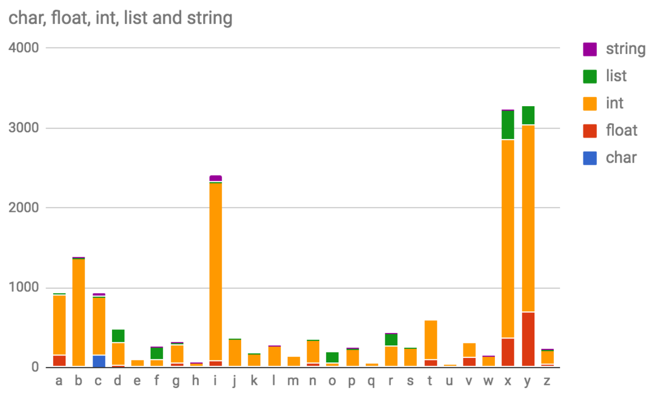


Fig. 8. Inferred types for variables of one letter

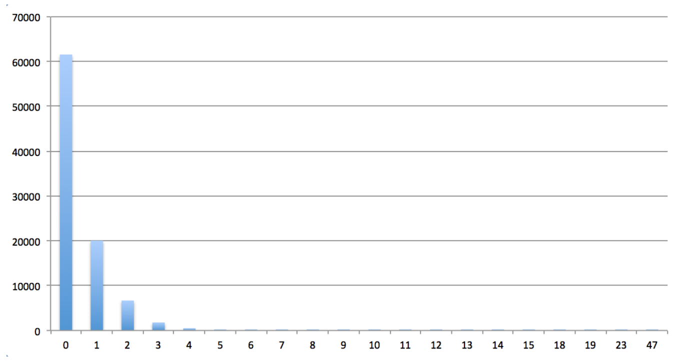


Fig. 9. Number of spaces in variable names

variable names. We have found that many introductory Scratch programming materials demonstrate the use of space free variables, and that children—and adults—that already have programming experience deem the use of spaces in variables as non-natural, even though arguable 'number of apples' is more natural that 'nApples'.

B. *Use of non-letter variable names*

In addition to spaces in variable names, Scratch even allows the use of numbers and even floating point numbers as variables. We found 718 projects with integer variable names and 19 with floating point names. While their use is rare, we manually examined some projects and numbers are used in interesting and clever ways.

here we can show the tic tac toe example

VII. Discussion

VIII. Conclusions

IX. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] K. Brennan, C. Balch, and M. Chung, *CREATIVE COMPUTING*. Harvard Graduate School of Education, 2014.

[2] G. Beniamini, S. Gingichashvili, A. Klein-Orbach, and D. G. Feitelson, "Meaningful identifier names: the case of single-letter variables," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, G. Scanniello, D. Lo, and A. Serebrenik, Eds. IEEE / ACM, 2017, pp. 45–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=3101421

[3] K. R. Gabriel, "Simultaneous test procedures—some theory of multiple comparisons," *The Annals Mathematical Statistics*, vol. 40, no. 1, pp. 224–250, 1969.

[4] D. W. Zimmerman and B. D. Zumbo, "Parametric alternatives to the Student t test under violation of normality and homogeneity of variance," *Perceptual and Motor Skills*, vol. 74, no. 3(1), pp. 835–844, 1992.

[5] F. Konietschke, L. A. Hothorn, and E. Brunner, "Rank-based multiple test procedures and simultaneous confidence intervals," *Electronic Journal of Statistics*, vol. 6, pp. 738–759, 2012.

[6] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload - A case study of the gnome ecosystem community," *Empirical Software Engineering*, vol. 19, no. 4, pp. 955–1008, 2014. [Online]. Available: https://doi.org/10.1007/s10664-013-9244-1

[7] B. Vasilescu, A. Capiluppi, and A. Serebrenik, "Gender, representation and online participation: A quantitative study," *Interacting with Computers*, vol. 26, no. 5, pp. 488–511, 2014. [Online]. Available: https://doi.org/10.1093/iwc/iwt047

[8] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information & Software Technology*, vol. 74, pp. 204–218, 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.01.004