FLAG LAB

Dissertation

Code as Art, Art as Code

The SCuLPT Programing Language

Nicolás Londoño Cuellar

June 3, 2025

This thesis is submitted in partial fulfilment of the requirements for a degree of Undergraduate in Systems and Computing Engineering.

Thesis Committee:

Prof. Nicolás Cardozo (Promotor) Universidad de los Andes, Colombia

Code as Art, Art as Code

The SCuLPT Programing Language

Systems and Computing Engineering Department

FLAG lab

Faculty of Engineering

Universidad de los Andes

Bogotá, Colombia

This manuscript has been set with the help of TeXShop and PDFLaTeX (with BibTeX support). Bugs have been tracked down in the text and squashed thanks to *Bugs in Writing* by Dupré [3] and *Elements of Style* by Strunk and White [5].

Thanks to everyone who made this possible and believed in the power of human computation.

Abstract

Everyone should have the opportunity to interact with code just as the average programmer does, no matter the condition or background they may have. The Simple Cubic Language for Programming Tasks (SCuLPT) is a programming language and artistic ecosystem designed to challenge the status quo of programming languages and computational thinking. SCuLPT disrupts the standard typewriter as the default interface for coding and programming languages with an intuitive, 3D interface and its specification. This is done by creating programming and artistic constructs that become tangible objects such that, when composed, generate computable programs and art pieces. Along the language, we provide a set of tools aimed to make art a tool for coding and programming an art form. Also included in the ecosystem exists the Simple Cubic Language for Programming Tasks Environment and Runtime (SCuLPTER) to run and test programs before assembling the final sculpture. The constructed language is a fully capable, Turing complete language that serves as a creative outlet as well. Using said tools, we perform an empirical validation of the experience with novices and expert programmers alike. The results show programming proficiency in solving small computational programs across several populations, and some really neat art!

Acknowledgements

Having so many to thank in such a small space is daunting, I do not want to skip anyone, nonetheless, I'll try to be succinct.

First and foremost, I would like to thank my advisor, Nicolas Cardozo, for listening to a weird idea from an even weirder student. His guidance has been incredible, not only in this work, but also in what follows from it.

I also thank the members of the FLAGLab. They gave me a space to talk about printed plastic pieces and esoteric programming languages. Your feedback have been invaluable in shaping this work and the language's technicalities. Making SCuLPT out of 3D printed pieces was a monumental task my poor Ender printer was simply incapable of, and I would like to thank the people at CREA and the guys running the department's 3D printers for their help in making this possible.

I will probably never be as good as a designer as my sister, Sara, but I will always be grateful for her help in making the SCuLPT visual language look amazing and work as well as they did. I also owe a lot to my mother, who has always supported me in my endeavours, even when they seem a bit out there.

I am not very good at organizing in my schedules. I would like to thank my partner, who has been there to mindlessly remind me of deadlines and to keep me on track by simply asking for updates. Your selfless interest on the project has been a great help in keeping me focused and on track. I am not sure how you put up with me, but I am glad you do.

Finally would like to thank my friends for putting up with me and my shenanigans, and for always being there to help me out when I needed it. Always at the ready to make me crack a smile or two when times got rough and my hands numb from hand-sanding every printed piece of SCuLPT.

Contents

List of Figures

Chapter 1

Introduction

The typewriter paradigm, through the keyboard and screen, has been the reigning interface to materialize programs into code since the beginning of programming as we know it today.

Its interface design lack what is needed to support every user. Naturally, users are not created equally. Prior work in the user experience and human-interface integration show reports in which technology and its designs can discriminate against users due to several factors, such as different physical or cognitive capacities, race and gender [4]. Moreover, programming has a steep learning curve and a high entry level [2]. These factors make programming languages and their interfaces inaccessible to a large portion of the population, disincentivizing them from learning to program and interact with code in the first place. Different approaches to computing are needed to lower the entry threshold and make computing more approachable to everyone.

Current programming languages are based on the same principles that we have used since the 50s. The same principles that were designed to be used with a typewriter and a flat surface in which to write. To lower the entry threshold of programming and computing, more approachable interfaces and methods are required. New fields in computing are emerging with different intents but still the same interfaces [6]. In order to explore new methods of computing, we must look further ahead than the tools we currently have and revisit how we approach code in general.

To do so, we must shift attention to other mediums that can be used for computation, yet serve a complete different purpose. Such path could lie into the realm of art. Using coding tools to create art is not a new concept at all. Many artists today embrace code in their practice, creating tools and environments to create interesting pieces of work. For example, Sonic Pi is

an entire coding suite built over SuperCollider sound synthesis server that allows users to create music using code with a superset of Ruby tailored for live coding performances and initially conceived as a programming teaching tool for children in England [1].

Creative computing is a field that has been explored for decades, with many artists and programmers creating tools and environments to create art using code. We may able to harness some of the principles of creative computing to shape new paradigms of computing that are more accessible and approachable to a wider audience. SCuLPT is an artistic sandbox with rules that ensure correct computation, shifting the focus away from computation, yet relying heavily on it to create physical sculptures that, by design, compile to valid programs. Alongside the physical sculptures, the SCuLPT ecosystem includes an environment and runtime to write, test and run programs before assembling the final sculpture.

1.1 Objectives

In the creation of SCuLPT we posit two objectives. First, we require to build a fully capable physical programming language addressing the concerns described previously about the state of programming, computation and programming languages. SCuLPT is designed to be a visual and physical oriented programming language, with a heavy focus on the ease of use, computational correctness and aesthetics. SCuLPT is designed to be a Turing complete language with a very small instruction set, allowing users to create any program they desire.

Second, we aim to evaluate the language in two main aspects. First, we intend to evaluate the ease of the language to foster computational thinking without the intrinsic difficulties of current programming languages imposed by the current interfaces. This is done by evaluating the performance of users with different backgrounds and levels of experience in programming, from novices to expert programmers, in solving small computational problems using SCuLPT and their overall experience with the language. Second, we evaluate the properties of SCuLPT as a paradigm that breaks existing preconceptions about programming languages and moves beyond the typewriter interface by leveraging artistic creativity as an alternative. This is done by letting users create pieces freely using SCuLPT, allowing them to explore the language and its components without any pressure to complete any tasks. This allows us to evaluate the overall experience of using SCuLPT as an artistic tool, as well as any suggestions for improvement from an aesthetic lens.

1.2 Methodology

1.2.1 Design and Implementation

SCuLPT is a visual and physical oriented programming language that aims to lower the entry threshold to programming by leveraging artistic creativity as an alternative to the traditional typewriter interface. SCuLPT as a whole is in fact two interconnected languages, one being the visual language and specification that defines the components from a strictly physical standpoint and the other being the programming language that is used to define the behaviour of each component. For the development of SCuLPT, we have followed a user-centred design approach, creating a visual language that is both usable and accessible to a wide range of users with the possibility of modifying the language to suit their needs. SCuLPT is focused on usability and accessibility is displayed by using simple but distinctive shapes and forms for each component on the language. This first design choice allows for users to easily identify and differentiate between different components of the language, making it trivial to convey meaning behind each shape while allowing people with any background to recognize each component foregoing knowledge, language or capacity barriers that arise from conventional programming languages.

Regarding the programming language, SCuLPT is designed to be a Turing complete language with a very small instruction set of only 13 instructions. By building structures and following the intuitive rules of the physical language, users can create any program they desire. The language is designed to be simple and easy to understand, with a focus on the visual representation of the program rather than the code itself. Code block are physically assembled together to create a program, with each block representing a different operation. Blocks are designed such that their connections are intuitive and easy to understand, with each block having a set of features that must be connected to other blocks in order to function correctly. This allows users to create programs without the need to understand complex syntax or semantics, making it accessible to users with little to no programming experience.

Regarding the language's aesthetics, SCuLPT is designed to be an artistic sandbox. The rules and specification for how shapes should look and feel are very loose. This allows users to tinker with the components, suiting them for their needs and desires. The only rules that must be followed only consider component connections and a set of features each block requires. Final

shape, size, material, colour or texture are completely open for anyone to modify without any impact on the program. This provides a sense of freedom and creativity that is not present in other programming languages, allowing users to express themselves through their code and create unique and personal pieces. This also implies that the language can be fitted to almost anyone's needs, allowing accessibility and customization for users with different physical or cognitive capacities or context where SCuLPT will be used.

SCuLPT current implementation used for testing features small 3D printed components in polylactic acid (PLA) with varying colours. The printed pieces follow the standard SCuLPT implementation with no modifications to the shapes. The components are connected using magnets and screw elbow joints, allowing for easy assembly and disassembly of the components, making it easy to create and modify programs on the fly.

SCuLPT has a double-edge sword, as it is strictly physical media, execution is tied to the interpreter, a human interpreter. Human interpretation is one of the main aspects of the language, yet humans are not perfect machines. This implies that correct computation is not guaranteed. Errors in the program's execution are bound to happen at some point and the language is not designed to be fault tolerant from a strict computation standpoint. To overcome this, SCuLPT also features the Simple Cubic Language for Programming Task's Environment and Runtime, SCuLPTER for short. Written in Scala, SCuLPTER is a transpiler written in Scala capable of taking a text representing a SCuLPT program and running it step by step to allow for debugging and testing of the program before it is physically assembled. SCuLPTER is designed to be lightweight and portable, allowing it to run in any browser thanks to the Scala.js web framework. This makes easily distributable and accessible to users wanting to have a reliable method of interpreting their sculptures.

1.2.2 Validation

Regarding validation, we have performed a series of sessions with users from different backgrounds and levels of experience with programming. For our sessions, users ranged from novices with no prior programming experience to expert programmers with years of experience in the field. Users also varied in ages, ranging from secondary school students to graduate students and

professionals. Tests were also performed with users with little to no knowledge in programming, but rather in art related fields. The tests were designed to evaluate the usability and accessibility of SCuLPT, as well as the overall experience of using the language.

This sessions consisted of several moments, starting with a brief introduction (30-45 minutes) to the language and its components. This introduction was tailored to each group of users, with novices receiving a more in-depth explanation of the language and its physical components, while expert programmers received a more technical overview of the language and its computational features. Afterwards, users were given the opportunity to explore the language and its components (1 hour), with a focus on the physicality of the blocks and their connections. Growing familiarity with the language and its components was the main goal of this section, allowing users to explore the language and its components without any pressure to complete any tasks. Next, users were given a set of tasks to complete using SCuLPT, with the goal of evaluating the usability and accessibility of the language. Several sets of tasks were created varying in complexity and difficulty to accommodate the different levels of experience of the users. This tasks consisted of simple programs that could be created using the language, such as creating a simple loop to add numbers or conditional statements to break from such loops.

During this test phase (2 hours), users were allowed to ask questions and receive help from the facilitators, but were encouraged to complete the tasks on their own. User were also encouraged to explore the language and its components, with the goal of creating a program that was both functional and visually interesting. Finally, users were given a survey to evaluate their experience with SCuLPT, with a focus on the usability, accessibility of the language, and their overall programming knowledge. In the special case of underage students, the accompanying teachers were asked to fill the survey as well. The survey also contained questions focused towards the educators themselves, such as their overall reception of the language and its components, as well as their thoughts on the potential use of SCuLPT in their classrooms. The survey was designed to gather feedback on the overall experience of using SCuLPT, as well as any suggestions for improvement from a technical and aesthetic lens.

Evaluation of the results was done using a combination of qualitative and quantitative methods. Qualitative methods included user interviews and surveys, while quantitative methods included measuring the time taken to complete each task and the number of errors made during the

process. Interviews were conducted after the tests to gather feedback on the overall experience of using SCuLPT with some individuals, as well as any suggestions for improvement from a technical view. The results of the tests were analysed to determine the overall usability and accessibility of SCuLPT, as well as the overall experience of using the language. Additional tests were performed to evaluate the overall experience of using SCuLPT as an artistic tool, with users being asked to create a piece of art using the language. The results of these tests were also analysed to determine the overall experience of using SCuLPT as an artistic tool, as well as any suggestions for improvement from an aesthetic lens.

1.3 Results

1.3.1 Language Implementation

SCuLPT as a programming language is a functioning language capable of writing any program.

The language uses FILO (First In Last Out) stacks as its only data structure to store and manipulate data. Each stack is represented by a different shape other than the reserved shapes used by the instructions, and they are initialized on first call. Stacks can hold an arbitrarily large amount of rational numbers. Stacks allow the pop, push, move and duplicate operations. Numbers are defined by their literal counterparts, and the language supports basic arithmetic. Regarding their representation, stacks are represented by any shape that is not reserved for the instructions„ such as simple shapes like circles, triangles or squares or complex drawings. Anything goes.

Arithmetic operations are the only operations available in SCuLPT. Available operations are addition, subtraction, multiplication, comparison, modulo and division. These are performed in two ways:

- **As unary operator**: The first two elements of the stack given by parameter are popped and operated and pushing the result back into the stack.
- **As binary operator**: The first element of the stack given by parameter is popped and the second parameter, a number, are operated with the result of the first operation. The result is pushed back into the stack.

In cases where the arithmetic is not defined, such as division by zero or modulo by zero, a *nil* value is pushed into the stack. When a *nil* value is encountered on arithmetic operations, other than comparisons, the program will halt in an error state.

SCuLPT allows for flow control through the use of loops, conditional blocks and jump blocks. Loops are physically represented by a literal loop of connected blocks or a single block jump with a negative number as a parameter. The jump block is a special block that allows for the execution to continue at any point by moving the execution from the current block to another with the offset given by parameter. Conditional statements are represented by a question block using a stack value as a parameter. Positive values are considered true and evaluate the following

block, and negative values are considered false and do not evaluate the affected block, but rather skips the next instruction.

SCuLPT is based on the physical blocks. Blocks as three-dimensional objects are unwieldy to be described in text. Therefore, we have made several valid notations for SCuLPT programs. Every block is represented as a square with the shape of each block inside the square. The connections between blocks are represented by arrows between the squares. As a first example and introduction to the language itself, we have implemented a simple program that calculates the Fibonacci sequence in SCuLPT.

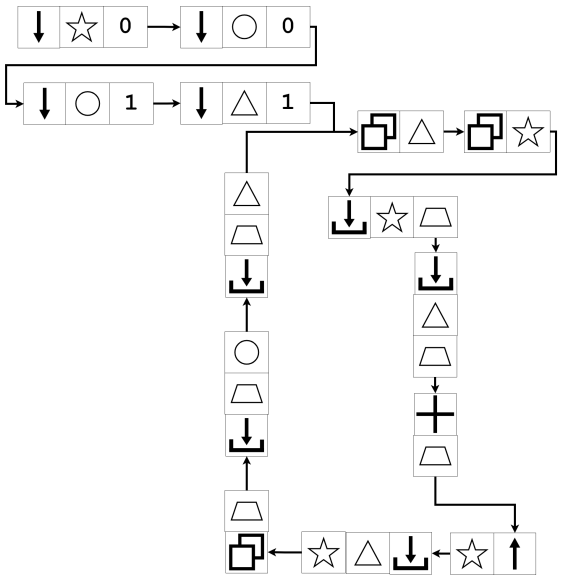

Figure 1.1: SCuLPT Fibonacci program

The program in Figure 1.1 fills the circle stack with the numbers of the Fibonacci sequence.

The second notation is a more traditional representation of code, which is used by the SCuLPTER interpreter, which will be explained afterwards. The program is written in a more traditional way, every block has a reserved name, connections are omitted. It reads as a list of instructions, somewhat similar to an assembly language. Limitations of this implementation are that the program is not visually represented, connections between blocks are not shown and physical loops cannot be represented, forcing the user to only use jump operations for execution control. Our Fibonacci implementation can be rewritten in SCuLPTER as follows.

Algorithm 1 Fibonacci sequence in SCuLPT

push 0 **star**
push 0 **circle**
push 1 **circle**
push 1 **triangle**
dup **triangle**
dup **star**
mov **star trapezoid**
mov **triangle trapezoid**
add **trapezoid**
pop **star**
mov **triangle star**
dup **trapezoid**
mov **trapezoid circle**
mov **trapezoid triangle**
jmp **-10**

Writing a Fibonacci program is a simple task in SCuLPT which gives us a good insight on how SCuLPT looks and feels, yet it is not a complete proof of the language's capabilities. The language is Turing complete and it is capable of writing any program that can be written in any other programming language. To validate this, we will simulate a Turing machine in SCuLPT. For this instance, an implementation of a 3-State Busy Beaver Turing machine will be used.

Its SCuLPTER implementation is as follows:

Algorithm 2 3-State Busy Beaver Turing machine in SCuLPT

push 0 **left**

push 0 **left**

push 0 **left**

push 0 **left**

push 0 **right**

push 0 **right**

push 0 **right**

push 0 **right**

push 0 **head**

// STATE A

push 1 **temp**

mov **temp head**

cmp **temp**

? **temp**

jmp 13 **// Go to A_1**

push 1 **head**

mov **right head**

mov **head left**

// STATE B

push 1 **temp**

mov **temp head**

cmp **temp**

? **temp**

jmp 9 **// Go to B_1**

push 1 **head**

mov **left head**

mov **head right**

jmp -16

// A_1

push 1 **head**

mov **left head**

mov **head right**

jmp 5 **// Go to C**

Algorithm 3 3-State Busy Beaver Turing machine in SCuLPT (continued)

```
// B_1
push 1 head
mov right head
mov head left
jmp −16 // Go to B
// STATE_C
push 1 temp
mov temp head
cmp temp
? temp
jmp 5 // Go to HALT
push 1 head
mov left head
mov head right
jmp −25 // Go to B
// HALT
push 1 head
```

This simulation of the Busy Beaver Turing machine is a simple program that runs for some time before halting. Since this simulation is of a Turing machine, it is capable of simulating any algorithm, thus proving that SCuLPT is Turing complete.

The complete implementation of SCuLPT is open source and is available on GitHub[1].

1.3.2 Interpreter Implementation

SCuLPTER is an interpreter for SCuLPT built in Scala that allows users to write and execute SCuLPT programs in a more traditional way. It leverages the Scala.js framework to provide a web-based interface for users. This makes the interpreter lightweight, distributable and easy to use without the need for any additional software. SCuLPTER features a simple text editor where a program can be written. It also features buttons to lex, parse and execute the program.

[1] `https://github.com/Darkoyd/SCuLPT`

Execution of the program is done in a step-by-step manner, allowing users to see the state of the program at each step. SCuLPTER code is open source and is also available on GitHub[2].

1.3.3 Validation

To validate SCuLPT, we have conducted a series of sessions with different users to test the language and its ecosystem. A total of 6 sessions were conducted with a total of 75 users, ranging from novices with no prior programming experience to expert programmers with years of experience in the field. The sessions were designed to evaluate the usability and accessibility of SCuLPT, as well as the overall user experience of using the language.

Regarding Usability

Usability was evaluated through questions in the survey given to users after the sessions, as well as through user interviews. The results of the usability questionnaire showed that users found SCuLPT, from a physical standpoint, to be easy to use and understand, with a majority of users reporting that they were able to use and understand the shapes and blocks of the language with relative ease. Among the users, especially the younger demographics, the physicality of the blocks was a key factor in their understanding of the language.

One concern that was raised by the researchers was that of overall piece integrity and the possibility of losing pieces. Continuous use of the blocks in a classroom setting could lead to pieces being lost or damaged, which could hinder the usability of the language. PLA is a material notorious for being a somewhat soft plastic prone to deformation and damage, especially with repeated use. As expected, some pieces were lost during the sessions, but the overall integrity of the blocks was maintained. The blocks were able to withstand the repeated use and manipulation by the users. The remaining pieces were still usable and intact, and the users were able to continue using the language without any issues.

These results shows that SCuLPT as a building block construct and visual language performs as expected, with users sharing overall positive feedback regarding the usability of the language.

[2] `https://github.com/Darkoyd/SCuLPTER`

Regarding Computation

Another section of the survey was dedicated to evaluate, from a more technical eye, the ease of use of SCuLPT as a programming language. Alongside the survey, users were given a set of programming tasks to complete using SCuLPT. The tasks were designed to test the users' understanding of the language and its capabilities, as well as their ability to write programs in SCuLPT. Throughout the sessions, about 50% of the total participants were able to complete the tasks. Further, per-session analysis shows interesting insights into the users' understanding of the language. The first session was conducted with a group of novice teenage users, who had no prior programming experience. These users were able to understand the basic concepts of SCuLPT yet they were not able to complete the tasks. The second session was conducted with a group of undergraduate students and expert programmers, who had years of experience in the field. Surprisingly, these users were also not able to complete the tasks, but they were able to understand the language and its capabilities. The next sessions were conducted with groups ranging from children to teenagers. Children evidently struggled with programming, yet outshines in the physical manipulation of the blocks. Teenagers, on the other hand, were able to understand the language and its capabilities, and they were able to complete the tasks in an average time of 1 hour and a half.

Results show that SCuLPT as a programming language can be easy to understand and use, yet it is not easy to write programs in SCuLPT. The language is capable of writing any program, but it requires a certain level of understanding of the language and its capabilities. This is a common issue with programming languages that SCuLPT tried to eliminate, but it is still present.

Regarding Aesthetics

Special sessions were held with a group of artists to test their overall experience with SCuLPT from their experience as artists. Results were surprisingly positive, with most users reporting that they enjoyed very much the experience of using SCuLPT. Participants expressed their appreciation for the physicality of the blocks and the ability to manipulate them in a tangible way. The physicality of the block "was quintessential to their experience". Participants also reported that the blocks were easy to understand and use, and that they enjoyed the process

of building their sculptures. When presented with the programming tasks, participants were able to easily understand the assignments, understand the block and their functionalities, but they were not able to complete any task. Nevertheless, they expressed overall satisfaction while trying and failing to complete the tasks. Participants reported that the experience of using SCuLPT was enjoyable and that they would like to use it again in the future. This last point lead to additional sessions with the same group of artists, as they were engaged with SCuLPT and wanted to explore it further. While trying and failing to complete the tasks, they used the blocks to create sculptures that were not intended to be functional, but rather artistic. Participants reported that they enjoyed the process of creating sculptures and that they found it to be a rewarding experience. Surprisingly, participants used the blocks in ways that were not intended from the start. Creating visually interesting constructs with this newfound freedom.

These results paint SCuLPT as a visually interesting, and even fun, medium for creative expression. Although this document has been focused on SCuLPT from a technical standpoint, it is important to note that SCuLPT was not only made to be a programming language, but also a medium for creative expression.

1.4 Conclusion and Future Work

1.4.1 Conclusion

SCuLPT is a visual and physical oriented programming language that aims to lower the entry threshold to programming by leveraging artistic creativity as an alternative to the traditional typewriter interface. We have presented the design and implementation of SCuLPT, a language that allows users to create physical sculptures that compile to valid programs as well as a functional environment and runtime to run and test programs before assembling the final sculpture. The design choices in SCuLPT make it possible for anyone to create programs without the need for prior programming knowledge, making it accessible and modifiable to a wide range of users.

Validation indicates that SCuLPT as an overall concept is well received by users, with many finding it intuitive and easy to use. The use of physical blocks to represent code constructs allows users to focus on the creative aspects of programming, rather than the technical details of syntax and semantics. Results show that SCuLPT shines in its ability to allow non-programmers to engage in code. Although the language is still in its early stages, it has the potential to be a valuable tool for artists, educators, and anyone interested in exploring the intersection of art and programming. SCuLPT is not intended to replace traditional programming languages, it is naturally unwieldy for serious projects, but functions to complement them by providing a new way of thinking about and creating programs.

1.4.2 Future Work

SCuLPT is still in its early stages and there are several areas for future work. First, we plan to conduct more extensive user studies to gather more data on the usability and accessibility of SCuLPT, as well as the overall experience of using the language. Second, we plan to explore the use of SCuLPT in different contexts and environments, such as in schools or workshops, to evaluate its effectiveness as a potential teaching tool. Third, we intend to enhance SCuLPTER to support more complex features, such as better error handling, a language server, syntax highlighting, and code completion. Also enhance SCuLPTER with interesting features aimed at artists, such as a visualizer to see the program as a sculpture before assembling it, exporting the

sculpture's blocks as 3D models for 3D printing and assembling and sculpture scanner to parse a sculpture and generate the corresponding program using images. Finally, we plan to explore the use of SCuLPT in different artistic contexts, such as in interactive installations or performances, to evaluate its potential as an artistic tool.

Bibliography

The references are sorted alphabetically by first author.

[1] Samuel Aaron, Alan F. Blackwell, and Pamela Burnard. The development of sonic pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming. *Journal of Music, Technology and Education*, 9(1):75–94, 2016. ISSN 1752-7074. DOI https://doi.org/10.1386/jmte.9.1.75_1. URL `https://intellectdiscover.com/content/journals/10.1386/jmte.9.1.75_1`.

[2] Yorah Bosse and Marco Aurélio Gerosa. Why is programming so difficult to learn? patterns of difficulties related to programming learning mid-stage. *SIGSOFT Softw. Eng. Notes*, 41 (6):1–6, jan 2017. ISSN 0163-5948. DOI 10.1145/3011286.3011301. URL `https://doi.org/10.1145/3011286.3011301`.

[3] Lyn Dupré. *BUGS in Writing: A Guide to Debugging Your Prose*. Revised edition, 1998. ISBN 0-201-37921-X.

[4] Amy Ko. How my broken elbow made the ableism of computer programming personal. *Nature*, September 2023. ISSN 0028-0836. DOI 10.1038/d41586-023-02885-y. URL `https://doi.org/10.1038/d41586-023-02885-y`.

[5] William Strunk and E.B. White. *The Elements of Style*. Longman, fourth edition, 2000. ISBN 0-205-30902-X.

[6] Hongji Yang and Lu Zhang. Promoting creative computing: origin, scope, research and applications. *Digital Communications and Networks*, 2(2):84–91, 2016. ISSN 2352-8648. DOI https://doi.org/10.1016/j.dcan.2016.02.001. URL `https://www.sciencedirect.com/science/article/pii/S2352864816000043`.

Acronyms