Bite Lang

Name: Kevin Liu

Advisor: Doosan, Baik

Presentation Date: December 2024

Graduation Date: December 2024

GITHUB LINK: https://github.com/Kevin27954/Pace_Honor_Thesis

Abstract

More and more high school students are now seeking to major in computer science. Education in programming is varied due to teaching methods, such as being taught in either Visual Programming or Textual Programming. The difference between most languages is that they are designed for production rather than education, creating a gap in tools tailored for novice learners. To solve this problem, I developed a new programming language dedicated to educational purposes with features like a gamified environment to improve learning retention. The future of this project is to enhance the features that will help students improve engagement and accessibility.

Table of Contents

Introduction

Throughout the past few years, more and more people are seeking to achieve results and hit it big in the computer science field. Since the start of 2012, the number of people pursuing a Computer Science degree or related field has increased yearly. From 2012 to 2022, it has increased by a massive 113%, from 50,961 to 108,503. In other fields of study, only a few in recent years have achieved similar growth, with the closest being health professions, with a 46% increase from 2012 to 2022 [12]. Without a doubt, following the trend, the number of people deciding on a path of CS during their secondary education will follow a similar trend.

For these new students to excel in CS, they need to understand how programming works, core concepts, and how to read programs and apply these concepts. A few languages have been aimed to help students learn programming early, like Blocky or Scratch, with features that allow them to code without the complications of variable declarations, functions, and intimidation of working with the command line. Python is another language often taught due to its simplicity compared to most other languages used in the industry, alongside Java, C, and C++ [14].

However, Scratch lacks the depth that is required for advanced learning, making it not suitable for middle to high-school students. On the other hand, languages like Python, though simple, are primarily designed for professional uses and can have a syntax that might be confusing or intimidating for students new to programming. This leaves a gap in educational tools specifically designed for students who are ready for more complex concepts but still need some form of user-friendly and learning-focused environment. This thesis aims to bridge the gap by creating a language aimed at middle to high-school adolescents. It will feature an easy-to-learn syntax and a gamified environment. Gamified education has often been successful in improving students' learning experience [15], so I plan on implementing a gamified

experience to teach the basics of the language and its features and understand the core data structures that will be common throughout their careers. This will be done within the terminal display itself, so the users will become familiar with the terminal environment, as it will be one of the most used tools in their careers. Giving students an early education in programming has shown positive effects in college [13], outperforming other students.

History

Programming languages have been around for centuries, dating back to 1849. It started with Babbage's difference machine and Lovelace, a British aristocrat who noticed the potential of his machine. She realized that she could represent the numbers as more than just numbers, leading to the first steps of language design [1]. The first programming language, Plankalkul, was invented in the 1940s by Konrad Zuse. It allowed engineers to store code and perform repetitive tasks much more efficiently. More efficient programming languages followed suit, with the first assembly language invented by Kathleen Hylda Valerie Booth in 1947. As the years went on and technology advanced, the types of created languages exploded, like ALGOL, which became the basis for the languages we know today: C, Java, and C++ [1].

The idea of programming languages invented for educational purposes only began in 1964 with the invention of BASIC, followed by Pascal in 1970 and Small Talk in 1972. BASIC (Beginners All-Purpose Symbolic Instruction Code) was invented to provide computer access to students [1]. It is now owned by Microsoft and is known as Visual Basic Classic. The Pascal language was intended to teach structured languages to students about good programming habits. Lastly, Small talk is object-oriented programming (OOP) made for educational purposes. These

languages share the same commonalities of being high-level and attempting to extract away the complications of machine code at the time.

As the years passed, Python, created in 1991, became popular because of its simplicity and choices to not focus heavily on OOP as a student's first CS course [2]. Many other languages, like C, Java, and C++, all of which were most commonly taught in classes CS1 and CS2 in the past, are being overtaken by Python. Over the years, it has become one of the top languages in the education environment taught in colleges [2]. Education-based programming languages made another leap forward by introducing advanced computer graphical interfaces. Visual Basic Classic was one of the first to launch and gain popularity with its simplistic user interface. However, it has not been maintained in its original form in recent years. Later, Scratch, created by MIT in 2007, soon exploded in popularity, with well over 100 million projects developed and shared by Scratch users, targeting children in primary schools.

Literature Review

Comparative Case Study

Block-based programming, Scratch, Block, etc., and Python are two popular educational languages for beginners, with one being the language for children and the other quickly gaining popularity in recent years [2]. Python has a simple syntax and is user-friendly to teach compared to other common text-based languages [4]. Visual programming languages like Scratch and Blocky can help students reduce syntax errors and cognitive loads on their learning [4].

Scratch's ease of use allowed many students new to programming to create programs quickly despite their lack of experience and sometimes educational background [3]. Scratch is also able to overcome language barriers when taught to students in other countries, like Africa

[3]. This is largely thanks to their visual block-based interface and color-coded script categories, which allow the students to remember based on colors rather than words [3]. Scratch was able to reduce the cognitive load on many students since they did not need to care about the syntax, and thus found learning Scratch to be a much more enjoyable experience compared to other languages, such as Python [4]. This also allowed the students to focus more on the conceptual aspect of programming, resulting in a much better and more enjoyable time learning.

Scratch was also able to find itself integrated into other areas and not just programming; important skills like problem-solving when students needed to debug their code, and other subjects, such as artistic expressions when the students designed their own story, were also found. It has been found that educational potential can foster growth in younger children, as many students showed positive outcomes at the end of their education despite a lack of computer experience. Some even found more tools and resources that a computer can provide for other purposes [3].

However, due to its visual nature, Scratch has limits when teaching programming. Although it can remove the worry of syntax errors, it makes it more difficult to learn another programming language [4]. This is crucial since most software is written in textual-based programming languages, and Scratch limits the ability to transition to a text-based environment.

Python is one of the most commonly used programming languages due to its simplified syntax, ease of use, and cross-platform compatibility [2, 4]. Opting for tabs or spaces to identify code blocks, the Python design aims to have an orthogonal, explicit, and minimalist design that will result in readable code. Its simplicity made it much easier to teach and faster to write example codes to teach the students [8]. There aren’t many setups for Python, and anything

taught can be written into code by the student almost immediately. As a result, they are also commonly used in educational settings [4].

Even though the syntax is simpler than other languages like Java, C, and C++, which are also often taught at school [2], students still need help when it comes to learning programming. The students commonly need help with syntax errors and writing programs efficiently due to the need to constantly recall the correct syntax [4]. This also becomes a problem when it comes to debugging since they won't be able to correctly read and understand their program, which would frequently lead to frustration [4]. When compared to visual languages, Python is the one with a higher ceiling.

However, in a case study [4], as students continue to learn Python, they find themselves developing the necessary skills to break down complex issues into more manageable parts as well as algorithmic thinking. This is an essential skill when it comes to programming. Many of the students also got more familiar with the syntax of Python, which they can transfer to other languages.

Challenges Students Face When Learning

Learning programming languages and concepts is not easy. It is almost akin to learning a new variation of English with new keywords, grammar, and meanings. Students often make many mistakes during the initial stages of learning a programming language. These common mistakes can be categorized into three parts: syntactic knowledge, conceptual knowledge, and strategic knowledge [6].

The students struggle with syntax errors such as missing parentheses, misuse of semicolons, and malformed expressions, which are common errors often detected by IDEs [4, 6].

Among these errors, there were also times when the students got confused over symbol meanings, such as = and ==. These are some of the most common mistakes that students face initially [6].

With conceptual knowledge, the students also struggled when it came to understanding the idea of variables, control flows, loops, functions and methods, and the meaning of a built-in word. This was also influenced by the lack of knowledge about the syntax of the language [4]. Some students thought that a variable could store multiple values simultaneously, misinterpreting loops and conditionals and believing an English meaning of the word compared to its actual purpose in the language [6].

Lastly, strategic knowledge. Students oftentimes struggle when it comes to applying their understanding to their programs, during planning, writing, and debugging [4, 6]. The students also show difficulty when it comes to higher-level problems due to the lack of programming strategies or patterns shown to them. Any attempts at solving resulted in an ineffective problem-solving approach. The students who struggled with syntactic knowledge had a much more difficult time when it came to strategic knowledge [6].

The study attempts to address some of the concerns and ways to improve the student's learning experience [6]. One of the things that will be crucial when it comes to improving the students is learning how to read and understand the code [6]. It is an important skill to have when planning and debugging code. The study recommends giving well-designed examples to students, explicit teaching or programming strategies, and also visualization tools. The examples and teachings can give the student a role model to look back to, and visualization can give the student the ability to understand the flow of the program by seeing how the code is executed step by step [6].

Gamification in Education

As research into educational methods progresses, gamification has emerged as a promising approach, particularly for teaching younger generations. A study was conducted to see the effects of teaching through a challenged-based gamification [5]. It consisted of 365 students from the Business Administration and School of Electrical and Computer Engineering (ECE). The study broke the students into four groups: Control, Reading, Gamification, Gamification & Reading. The students would be scored based on a final evaluation form, which consisted of 30 closed-ended questions that were based on the lecture material to assess the student's understanding of the topics taught during the experiment [5]. The study found that students exposed to gamified learning environments scored substantially higher on assessments compared to those in traditional learning settings. Notably, when gamification was combined with traditional reading assignments, students achieved the highest scores, scoring 58.05 out of 100, nearly doubling the results of the control group, scoring 36.13 out of 100 [5].

Similarly, apps like Duolingo, which debuted in 2012 as a way for people to learn other languages easily from their phones, have used a challenged-based gamification approach in teaching their users new languages. Duolingo by itself is already successful as it had 34 million daily active users in the first quarter of 2024 [11]. When they are paired in a classroom environment, the effect of gamification takes on a greater effect. Many of the features in Duolingo, like hearts, points, currency, and leaderboards, have increased the student's motivation and engagement and allowed for longer motivation [10]. From the study [10], many of the students found Duolingo to be a much more enjoyable learning experience compared to in-class literature when it came to learning newer languages [10].

Gamification in education represents a significant advancement in engaging students and enhancing learning outcomes. The evidence from studies and practical applications like Duolingo underscores its potential to transform traditional educational approaches, making learning more interactive and enjoyable for students.

Designing the Language

Designing a language is not a simple task, especially when it comes to making one for students. In their panel discussion on "Designing the Next Educational Programming Language," Black et al. [7] outlined a comprehensive set of principles for such a task. These principles will serve as guidelines for the type of language I aim to develop, providing a framework for balancing educational value with practical usability.

The language will follow the Lox language by Robert Nystrom, which is a scripting language similar to well-known languages like Python and JavaScript. As a result, it will be simple so that anyone can pick it up just as easily. It will utilize syntax inspired by Lua, which is very English-like and has the complete word rather than an abbreviated version. The grammar is based on the Lox language's structure, with some modifications.

Since the language is meant for beginners to learn some early and basic programming concepts, it will have no high-level features such as concurrency, classes, graphics, and inheritance to avoid overflowing the student with information. Most of the functionality of the language will be common across other text-based languages that are commonly used in the industry.

and	or	true	false	none	if	then	end
else	function	struct	for	while	do	let	return

Figure 1

There will also be a small gamified environment where the students can learn how a language works based on examples and exercises. Quizzes will be given at intervals to ensure the student understands the language correctly. This aims to familiarize the students with the syntax of the language and the concepts they will face commonly as they progress.

Process

For this project, I plan on using the book *Crating Interpretors* by Rober Nystrom as a primary resource for building my language, with some adaptions to account for differences between Rust and the language used in the book, Java and C. However, translating his concepts into Rust introduces some unique problems, particularly memory management, which I will talk about later.

One of the key parts of any language is its syntax. The keywords were designed to be as simple as possible, taking inspiration from Lua and Python. For instance, rather than using semicolons, or braces to separate code or blocks, we use the keywords do and end, and use newline character to serve as a line terminator.

Following Nystrom's structure, the project will be split into two parts. The first half focuses on implementing a recursive decent interpreter and will be used as a learning exercise to understand the concept of making an interpreter, especially in Rust. This will serve as the foundation for the second phase, which involves building a single-pass, bytecode-based compiler. The bytecode implementation will be the version presented as it is more efficient than the recursive descent.

Recursive Decent

The first half of the book introduces recursive descent parsing- a top-down approach where the parser descends through the parse tree and evaluates them. This gave me a great opportunity to familiarize myself with Rust's features while implementing the entire process from scanning to interpreting.

The process starts first with the scanner which takes in input categorizes it into token types and stores it in a Token struct. Each instance of a struct Token contains the token type, the line number for outputting error, the runtime Literal value, and its representation for printing. The literal is represented as an Option<Literal> so it can also handle variable names, operators, and other tokens without values during runtimes. Where Some (Literal) is for values and None is for tokens without values.

```
struct Token {
    token_type: TokenType,
    lexeme: String,
    literal: Option<Literal>,
    line: u32,
}
```

Figure 2

In the parsing phase, it receives a vector of all the tokens from the Scanner and checks it against the grammar rule, organizing the tokens into a parse tree. These parse trees are represented using enums with tuple structures in Rust, however, in retrospect, enums with structs could have provided more clarity. I wanted to utilize the match statements that Rust had, which allowed me to check for all possible enumerations during interpretation, enhance error checking, and ensure all types were handled.

Next, the interpreting phase goes through each parse tree, executing the instruction and managing the variable states in the runtime environment. The runtime environment is a link list

of hashmaps, each node corresponding to a scope, and the hashmap stores the variable names and their values. This setup allows inner scope to refer to outer scope variables unless shadowed.

To be able to handle static scoping, Nystrom proposes an intermediary step called resolver [16]. The resolver inspects the parse tree before runtime and determines the scope of each variable reference it belongs to. This information is stored in a hashmap that the interpreter can reference when interpreting the parse tree. This step allows for optimization lookup during runtime and guarantees the variable references are valid based on the static scoping rules.

Bytecode

For the bytecode implementation, Nystrom introduces a single-pass compiler, where the compiler and the scanner are both done in the same step before execution by the Virtual Machine (VM) [16]. This improves the overhead of having two phases and managing the memory when it comes to the compilation process.

To achieve the single-pass compilation, Nystrom uses Pratt's Parsing algorithm, which allows us to take in a stream of input in O(N) time complexity. Pratt's Parsing evaluates the expressions as it reads more tokens and determines the operation and the context on the fly.

In bytecode, the VM runs bytecode instructions. As a result, we only need to have representation for the runtime values in the VM, so this eliminates the need for a parse tree. Numbers, for example, are represented as f64, simplifying the implementation by treating both integers and floats as the same type. Other more complex data types, such as strings or functions, are allocated on the heap as Obj enums.

enum Value {
  Number(f64),
  Boolean(bool),
	enum Obj {
	  String(Rc<RefCell<StrObj>>),
	  Function(Rc<RefCell<FunctionObj>>),

None, Obj(Obj), }	NativeFn(Rc<RefCell<NativeFn>>), Structs(Rc<RefCell<Structs>>), Instance(Rc<RefCell<StructsInstance>>), }

Figure 3

During runtime, the VM only needs to execute the instructions. All the scoping and variable references are already determined during compilation. This allows the VM to avoid any complex lookups and focus solely on the execution of the instructions, improving the efficiency compared to recursive decent.

At startup, the VM loads the main function by pushing the main function onto the CallFrame and the stack, which represents the start of the program. Local variables are stored on the stack in the order they are declared and defined. This avoids the need for a linked list and keeps the program simple. When we need to modify or get a local variable, we emit a bytecode OpSetLocal(idx) or OpGetLocal(idx), where the idx is the location of the local variable value on the stack relative to the scope location, tracked by the CallFrame struct.

Garbage Collection is vital in all languages to manage memory. Initially, my idea was to rely on Rust's memory safety features, namely the borrow checker and Rc<T> (Reference Counting) pointers. This removed the need for a heap that I needed to manually manage and allowed for automatic memory management. However, this approach conflicted with the implementation, as some of the objects were also stored in the vector maintained by each function. This resulted in objects being retained in memory for the entire duration of the program. Consequently, implementing the garbage collector became much more difficult in the limited time frame. Without a proper garbage collector, the current implementation lacks any memory management when it comes to non-primitive object values.

Gamified Environment

This environment will run in the terminal and be built into the language command-line interface (CLI). By creating the practice exercises in the CLI, users can learn the language more efficiently without wasting time searching for external resources.

To create a more game-like experience, the environment will include a progress tracker for users to track the next pop quiz which serves as a challenge where no hints will be provided. This quiz tests the user's ability to write a program that will execute successfully and demonstrate their knowledge thus far. Error messages will be enhanced with color-coded outputs between errors and successful program runs. Throughout the environment, users can access hints when difficulties are encountered, helping progression without encountering frustrations.

The learning curve will be incremental, targeting mainly beginners with gradual increases in difficulty. There will be an initial setup stage for the user to experience a test run to familiarize themselves with how to pass or fail a stage.

Behind the scenes, the environment operates by spawning two threads: one for reading user input and another for file-related tasks. The file thread checks if the file has been modified every 200 milliseconds and runs the command to compile and run the file. The status code returned by the compilation determines if the user has written valid code and if they passed the stage. All the stage info is initialized at the start of the program, allowing progress tracking in runtime without needing to store more information in storage. The user input thread continuously reads user input and either quits if the user types "quit" or gives hints when "hint" is typed.

To avoid any concurrency problems, the threads only communicate when a user quits the program. The project utilizes Rust's built-in type safety, like Arc<T> pointers and Mutex<T>

pointers to ensure thread-safety operations and avoid any type of race conditions.

Limitations and Future Work

The current progress of this project can only be considered a demo. Due to time constraints, many features originally planned had to be excluded from the final product. Additionally, my limited experience with Rust posed many challenges at the beginning of the project, making bad decisions and practices in rust while developing the project.

Future work will focus on a complete rewrite of the system, with improvements such as garbage collection using an arena memory management strategy for efficiency and expanding the learning environment to incorporate a web-based version and a more interactive terminal environment to enhance accessibility and user engagement. Further integrations can be made with the learning environment and the language itself, such as unlocking advanced features and concepts only when the user has completed specific stages, ensuring that they have mastered key concepts to progress.

Conclusion

This thesis aims to develop a more suitable language for students who want to become more familiar with the CS field in a gamified learning environment. Integrating practice exercises and real-time feedback, the learning environment aims to enhance the learning experience and demystify programming concepts, making learning programming more engaging, especially for beginners.

This project is an initial step towards a new generation of coding education and has the potential for further improvement with features like AI-driven learning assistance, more customizability to the educator and the student using the language, and adaptive learning contents. These features can support a more personalized and inclusive learning experience.

Ultimately, this work aims to contribute to the growing field of education technology, showing how gamification and design aimed at students can transform how students approach learning programming and fundamentals. This project demonstrates the potential for technology to make programming more effective and engaging.

EBNF Grammar

Program

```
Program -> Declaration | Program EoF

EoF -> “\0”
```

Declaration

```
Declaration -> StructDecl | FuncDecl | VarDecl | Statements

StructDecl -> "struct" Identifier "{" Parameters "}"

FuncDecl -> "function" Function "\n"

VarDecl -> "let" Identifier ("=" Expression)? "\n"

Function -> Identifier "(" Parameters ")" BlockStmt
Parameters -> Identifier | Parameters "," Identifier
```

Statements

```
Statements -> ExprStmt | IfStmt | ForStmt | WhileStmt | ReturnStmt | BlockStmt

ExprStmt -> Expression + "\n"

IfStmt -> "if" Expression "then" Declaration ( "else" Declaration )? "end"

ForStmt -> "for" "(" ForStmtBlockOne + ForStmtBlockTwo + ForStmtBlockThree ")"
Statements

ForStmtBlockOne -> VarDecl | Expression | ","

ForStmtBlockTwo -> Expression "," | ","

ForStmtBlockThree -> Expression | ""
```

```
WhileStmt -> "while" "(" Expression ")" Statements

ReturnStmt -> "return" (Expression)? "\n"

BlockStmt -> "do" "\n" Declaration "end" "\n"
```

Expressions

```
Expression -> Assignment

Assignment -> ( Call "." Identifier | Identifier ) "=" ( Assignment | Logical-Or )

Logical-Or -> Logical-And | Logical-Or "or" Logical-And

Logical-And -> Equality | Logical-And "and" Equality

Equality -> Comparison | Equality ( "==" | "!=" ) Comparison

Comparison -> Term | Comparison ( "<" | ">" | "<=" | ">=" ) Term

Term -> Factor | Term ( "+" | "-" ) Factor

Factor -> Unary | Factor ( "*" | "/" ) Unary

Unary -> ( "!" | "-" ) (Call | Unary)

Call -> Primary ( "(" Arguments ")" | "." Identifier | "{" "}" )

Primary -> String | Number | Decimal | Boolean | None | "(" Expression ")" | Identifier

Arguments -> Expression | Arguments "," Expression
```

Basic Values

```
Identifier -> Alpha | Identifier ( Number | Alpha )

String -> "\"" <any character> "\""

Boolean -> true, false
```

Number -> (Integer | Float)

Float -> Integer "." Integer

Integer -> Integer Digit | Digit

Digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Alpha -> <"a" to "z"> | <"A" to "Z">
None -> “none”

Reference List

[1] J. Roller, “Coding from 1849 to 2022: A guide to the timeline of programming languages,” IEEE Computer Society, https://www.computer.org/publications/tech-news/insider-membership-news/timeline-of-programming-languages (accessed Oct. 23, 2024).

[2] R. M. Siegfried, K. G. Herbert-Berger, K. Leune, and J. P. Siegfried, Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning, https://research.leune.org/publications/ICCSE21-CS1CS2.pdf (accessed Nov. 1, 2024).

[3] A. Sikwebu and D. v. Greunen, "Starting from Scratch: Introducing Primary School Learners to Programming," 2020 IST-Africa Conference (IST-Africa), Kampala, Uganda, 2020, pp. 1-9. (accessed Nov. 1, 2024).

[4] A. Unal and F. Topu, "A Comparative Case Study to High School Students' Experiences in Programming Education: Python Editor versus Blockly Tool," e-Kafkas Eğitim Araştırmaları Dergisi, vol. 9, 2022, doi: 10.30900/kafkasegt.1053820 (accessed Nov. 1, 2024).

[5] N.-Z. Legaki, N. Xi, J. Hamari, K. Karpouzis, and V. Assimakopoulos, "The effect of challenge-based gamification on learning: An experiment in the context of statistics education," International Journal of Human-Computer Studies, vol. 144, 2020, Art. no. 102496, doi: 10.1016/j.ijhcs.2020.102496 (accessed Nov. 1, 2024).

[6] Y. Qian, J. Lehman, Inv. P. Yizhou QianPurdue University, and Inv. P. James LehmanPurdue University, “Students’ misconceptions and other difficulties in introductory

programming: A literature review: ACM Transactions on Computing Education: Vol 18, no 1," ACM Transactions on Computing Education, https://dl.acm.org/doi/10.1145/3077618 (accessed Nov. 1, 2024).

[7] A. Black, K. B. Bruce, and J. Noble, "Designing the next educational programming language," Designing the Next Educational Programming Language, http://gracelang.org/documents/panel.pdf (accessed Nov. 1, 2024).

[8] F. Georgatos, How applicable is Python as first computer language for teaching programming in a pre-university educational environment, from a teacher's point of view?, https://arxiv.org/ftp/arxiv/papers/0809/0809.1437.pdf (accessed Nov. 1, 2024).

[9] M. Shortt, S. Tilak, I. Kuznetcova, B. Martens, and B. Akinkuolie, "Gamification in mobile-assisted language learning: a systematic review of Duolingo literature from public release of 2012 to early 2020," Computer Assisted Language Learning, vol. 36, no. 3, pp. 517–554, 2021, doi: 10.1080/09588221.2021.1933540 (accessed Nov. 1, 2024).

[10] F. Azhima and A. Halim, The Integration of Duolingo in Classroom Setting: A Case Study of its Impact on English Language Learning, https://jurnal.usk.ac.id/riwayat/article/download/39326/21088 (accessed Nov. 1, 2024).

[11] D. Belevan and S. Dalsimer, "Duolingo reports 45% revenue growth and record profitability in first quarter 2024; raises full year guidance," Duolingo, Inc., https://investors.duolingo.com/news-releases/news-release-details/duolingo-reports-45-revenue-growth-and-record-profitability (accessed Nov. 1, 2024).

[12] Institute of Education Science, “Digest of Education Statistics,” National Center for Education Statistics (NCES) Home Page, a part of the U.S. Department of Education, https://nces.ed.gov/programs/digest/d23/tables/dt23_322.10.asp (accessed Aug. 13, 2024).

[13] J. Cárdenas-Cobo et al., “Using scratch to improve learning programming in college students: A positive experience from a non-weird country,” MDPI, https://www.mdpi.com/2079-9292/10/10/1180 (accessed Aug. 13, 2024).

[14] R. M. Siegfried, K. G. Herbert-Berger, K. Leune, and J. P. Siegfried, “Trends of commonly used programming languages in CS1 and CS2 learning: IEEE conference publication: IEEE Xplore,” ieeexplore.ieee.org, https://ieeexplore.ieee.org/document/9569444 (accessed Aug. 13, 2024).

[15] N.-Z. Legaki, N. Xi, J. Hamari, K. Karpouzis, and V. Assimakopoulos, “The effect of challenge-based gamification on Learning: An experiment in the context of statistics education,” International Journal of Human-Computer Studies, https://www.sciencedirect.com/science/article/pii/S1071581920300987 (accessed Aug. 13, 2024).

[16] R. Nystrom, *Crafting Interpreters*, 1st ed. Genever Benning, 2021.