Evy, a New Approach to Introductory Programming

Julia Ogris, MSc BBA

Abstract — Introductory programming faces a significant challenge: While block-based languages like Scratch provide an accessible entry point, the ultimate goal of programming education remains mastery of text-based languages. This creates a gap for learners transitioning from the intuitive visuals of blocks to the abstract syntax and, in many cases, the advanced concepts of modern, conventional programming languages. Evy addresses this gap with a novel approach to introductory programming.

At its core, Evy is a simple, text-based programming language designed for clarity, readability and learnability. With fewer special characters and advanced concepts than most conventional programming languages, it draws inspiration from beginner-friendly languages of the past like Basic and Pascal but also from modern block-based environments. Evy's minimalist syntax and small set of built-in functions reduce cognitive load and promote memorability, allowing learners to more readily access core computational and algorithmic thinking skills.

Beyond functionality, Evy recognizes the importance of engagement. Its small yet powerful set of built-in functions enable the creation of visually appealing graphics, interactive animations, and even games, fostering a stimulating learning environment that encourages independent exploration and experimentation. Accessibility remains paramount. Evy's web-based learning playground offers immediate access with no setup required, allowing exploration of varied examples and instant coding for beginners. However, for those seeking a deeper dive, Evy also integrates with preferred editors, terminals, and IDEs, accommodating diverse learning preferences.

Index Terms — computer science education, introductory programming, novice programmers, programming, programming environment, programming languages

I. Introduction

The landscape of introductory programming languages often lacks options tailored to the unique needs of beginners, especially children and teenagers. While block-based languages like Scratch or Microsoft's MakeCode offer intuitive environments [3, 4, 22] , transitioning to text-based languages like Python can be challenging [1, 2, 11, 24, 25, 26]. Conversely, conventional introductory languages like Python, C, Java, or Javascript often overwhelm beginners with complex syntax and advanced concepts [13 11, 17, 23]. This paper introduces Evy (https://evy.dev), a novel educational programming language designed to bridge this gap.

In the following we will explore the strength and weakness of various introductory programming languages, historical and present. We will also analyze the benefits and disadvantages of block-based versus text-based languages. Finally we will introduce the Evy language and show how it draws upon the strengths of various approaches to tackle the challenges of novice programming education.

II. Objectives

Evy is developed for STEM educators, researchers and most importantly, the next generation of coders, looking to bridge the gap between beginner-friendly block-based languages and powerful, conventional languages. It serves as a stepping stone, offering a minimal text-based language with a simple syntax and a small yet powerful set of built-in functions. Specifically designed for learnability, Evy prioritizes engaging experiences alongside core programming concepts, emphasizing computational thinking and "real" coding in a fun and accessible way.

While Evy the language has already been fully implemented, the project's aspirations extend further. We are actively working towards a well-researched learning platform and course framework that tackles head-on the shortcomings of conventional text-based programming language teaching. For this purpose, the project and its creators are seeking a funded research home, ideally at a university or research lab with expertise in educational technology.

III. Approach

This section explores the evolution of introductory programming, examining the merits and drawbacks of prominent approaches. Finally, it introduces Evy, a novel introductory language whose syntax, built-in functions, and environment are carefully designed for learnability, harnessing the strengths of existing approaches.

A. Historical Background

The difficulty of learning to program casts a long shadow, evidenced by high drop-out rates in introductory university courses (CS1). Some argue that programming is inherently challenging [17, 23], while others point to decades of successful instruction, even at the elementary level [13, 14, 16]. So, why the disparity? Why does programming persist as a perceived hurdle?

Luxton-Reilly [13] compellingly argues that the difficulty lies not in programming itself, but in flawed teaching methods and unrealistic expectations. We demand rapid assimilation of complex programming concepts while simultaneously expecting students to navigate intricate systems and tools. Crucially, we neglect to explicitly teach or assess these essential skills. Unrealistic expectations, a heavy workload, and the rapid ramp-up in complexity of programming concepts can lead to plagiarism, dropouts, and discourage students. This impact is particularly pronounced for those without prior programming experience, disproportionately affecting women and diversity in general [13].

Anecdotally, several text-based languages of the past had a narrower focus, with some explicitly designed for learning and teaching. Many older programmers fondly recall starting with Basic, which, despite limitations, offered a straightforward approach tailored for newcomers – even its name, *Beginner's* All-purpose Symbolic Instruction Code, was a testament to its purpose. Similarly, Pascal, with its stated design goal of being a "teaching" language [29], and its clear syntax, facilitated a smoother learning experience.

In a comprehensive analysis of introductory computer science courses, Sobral investigates the evolution of introductory programming languages by leveraging the extensive database of research papers available through Google Scholar [18]. The study examines publications mentioning "CS1," "introductory programming," and "novice programming" across various programming languages, spanning the years 1989 to 2018. Sobral's findings reveal a growing interest in the novice programming domain, with C, Java, and Python dominating the last two decades. Conversely, Pascal and Basic experienced a decline in prevalence, while JavaScript and Python demonstrate a notable upward trajectory[18], reflecting broader industry shifts [19]. Figure 1 presents a simplified depiction of these findings according to Sobral [18].

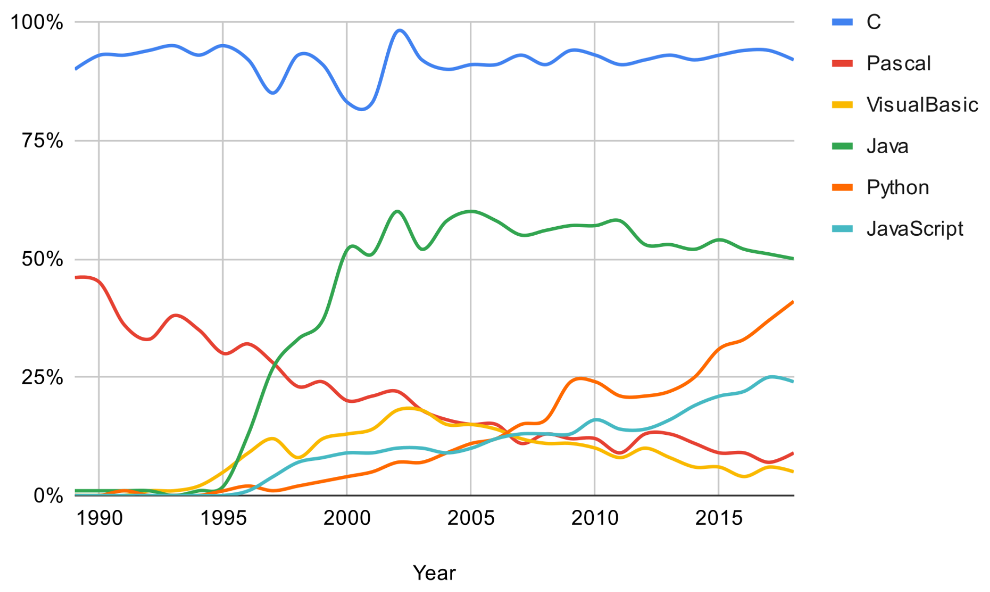


Figure 1: *Google scholar articles on introductory programming by programming language mentioned over all introductory programming articles*

B. Literature Review of Contemporary Approaches

Emerging in the early 2000s, block-based coding introduced a novel paradigm for teaching programming. This approach utilizes a visual building block metaphor, where commands are presented in logically organized categories, guiding learners on how and where to use them [3]. This eliminates the risk of syntax errors, making computer science more accessible and understandable for novice programmers [8, 20, 24]. The rise of graphical interfaces, the interactive potential of the web, and growing international mandates for coding education in schools all contributed to the popularity of block-based coding.

From Alice's pioneering role in the late 1990s to Scratch's thriving online community (see figure 2), several block-based languages have shaped introductory programming, including:

- **Alice**: Emerged in the late 1990s at Carnegie Mellon (https://www.alice.org/), evolved from a VR prototyping tool into a versatile block-based environment fostering creativity and storytelling [6, 22].
- **Scratch**: Released in 2007 by MIT (https://scratch.mit.edu/), Scratch revolutionized novice programming with its intuitive interface, engaging game and story creation features, and vibrant online community. It also offers Scratch Jr. for younger learners, solidifying its legacy as a leading platform [22].
- **Snap!**: Released in 2011 by UC Berkeley (https://snap.berkeley.edu/) Snap! shares similarities with Scratch but delves deeper with its expanded block library, first-class procedures, and list operations, catering to slightly more advanced learners [9].

- **MakeCode**: Released in 2017 by Microsoft (https://makecode.com), MakeCode empowers novices to explore robotics and microcontroller programming. [4]
- **Blockly**: Released in 2011 by Google (https://developers.google.com/blockly) Blockly is a block-based language and library that transpiles to Python, JavaScript and other modern text-based languages. It now serves as the backbone for Scratch and MakeCode. [21].

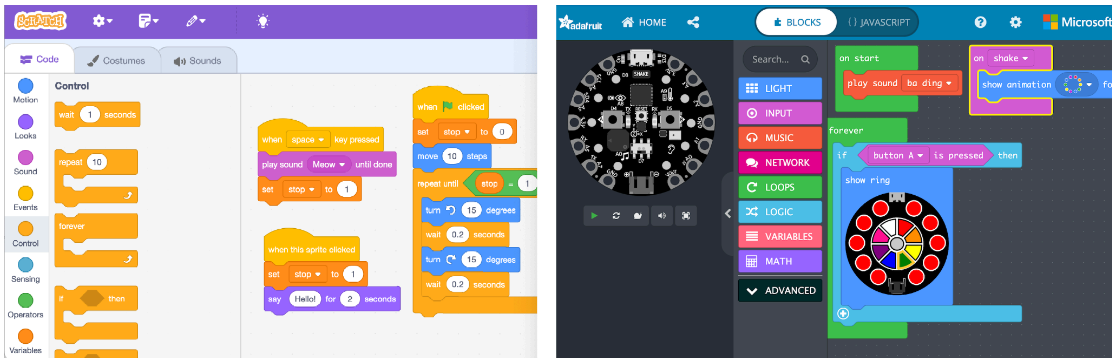
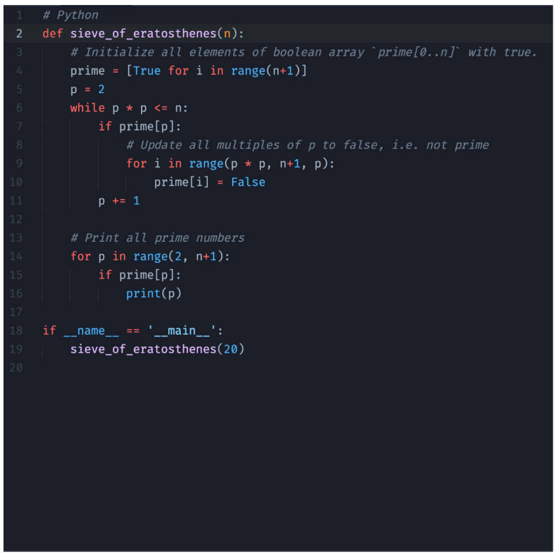


Figure 2: *A side-by-side comparison of programming environments: Scratch* (left) *and MakeCode with Adafruit Circuit Playground Microcontroller* (right).

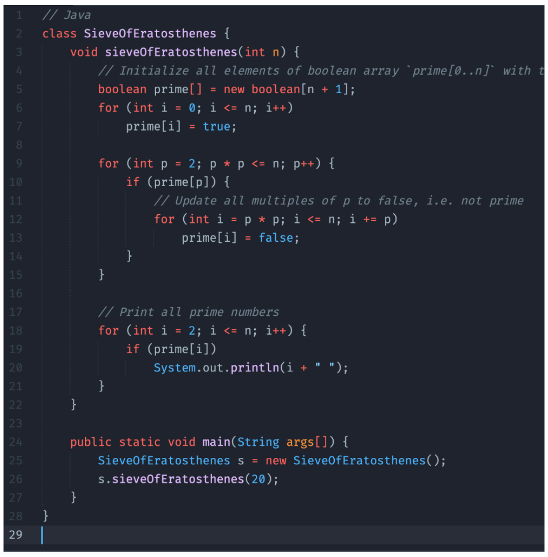
While block-based languages provide a user-friendly entry point for novice programmers, text-based programming remains the dominant approach in universities, many high schools and some primary schools. Its emphasis on syntax and fundamental programming principles prepares students for the challenges of advanced coding and equips them with the skills needed to tackle real-world projects [19, 30]. Figure 3 illustrates the implementation of the Sieve of Eratosthenes, a classic algorithm for prime number computation, in two of the most popular text-based introductory languages: Python and Java. These languages, along with JavaScript and C, also hold significant weight in open source development and industry demand [19]. Among them, Python stands out for its perceived ease of use and growing popularity. In fact, it is commonly used as an introductory language across Chinese, British, American and Australian high schools [7, 15, 19].



```python
# Python
def sieve_of_eratosthenes(n):
    # Initialize all elements of boolean array `prime[0..n]` with true.
    prime = [True for i in range(n+1)]
    p = 2
    while p * p <= n:
        if prime[p]:
            # Update all multiples of p to false, i.e. not prime
            for i in range(p * p, n+1, p):
                prime[i] = False
        p += 1

    # Print all prime numbers
    for p in range(2, n+1):
        if prime[p]:
            print(p)

if __name__ == '__main__':
    sieve_of_eratosthenes(20)
```



```java
// Java
class SieveOfEratosthenes {
    void sieveOfEratosthenes(int n) {
        // Initialize all elements of boolean array `prime[0..n]` with t
        boolean prime[] = new boolean[n + 1];
        for (int i = 0; i <= n; i++)
            prime[i] = true;

        for (int p = 2; p * p <= n; p++) {
            if (prime[p]) {
                // Update all multiples of p to false, i.e. not prime
                for (int i = p * p; i <= n; i += p)
                    prime[i] = false;
            }
        }

        // Print all prime numbers
        for (int i = 2; i <= n; i++) {
            if (prime[i])
                System.out.println(i + " ");
        }
    }

    public static void main(String args[]) {
        SieveOfEratosthenes s = new SieveOfEratosthenes();
        s.sieveOfEratosthenes(20);
    }
}
```

Figure 3: *Text-based programming: Python (left) and Java (right)*

C. Analysis

Text-based languages

Before diving into contemporary trends, let's take a quick look back at the historical landscape of introductory programming languages. Procedural languages like Pascal and C, alongside the early, unstructured languages like Basic, dominated the 80s, 90s, and early 2000s. Following their reign, Java rose to prominence as object-oriented programming gained traction, while Python and JavaScript emerged as versatile languages combining multiple paradigms. Notably, C has retained its dominance, likely due to its unwavering role in systems and embedded programming, where its performance and efficiency remain crucial.

While the dominant paradigms in both introductory programming and industry have shifted away from procedural languages, some research suggests a potential pedagogical advantage for these historically prevalent approaches. Notably, a study by Wiedenbeck et al [28] revealed that novice learners encountered significant difficulty with object-oriented (OO) programs compared to functionally equivalent procedural versions. While OO programming builds upon the fundamental concepts mastered in procedural languages, its additional features pose an initial hurdle for novice learners.

Following, a selection of dominant introductory programming languages is briefly reviewed:

- **C** (1972): This pioneer, designed by Dennis Ritchie, laid the foundation for modern languages with its procedural approach and static typing. Its influence in systems and embedded programming remains significant.
- **Java** (1995): James Gosling's object-oriented language transformed enterprise development with debatably robust features and platform independence. Today, it plays a major role in web applications and Android development.
- **JavaScript** (1995): Brendan Eich's creation transformed the web. Its multi-paradigm nature and dynamic typing make it essential for interactive web experiences and increasingly relevant for server-side programming.
- **Python** (1991): Guido van Rossum's brainchild rose to prominence with its arguably beginner-friendly syntax and versatility. Widely used in data science and machine learning, its popularity continues to soar.

These four prominent languages, all over 25 years old, have enjoyed the benefit of time to mature, refine, and accumulate extensive documentation and learning materials. However, they also carry the weight of their history, with legacy features and complexities that can challenge newcomers.

Python, one of the most popular modern programming languages, is often chosen as an introductory language due to its apparent simplicity and widespread popularity [5, 10]. While Python's syntax indeed boasts elegance and directness, it also enjoys the advantage of a more consistent "idiomatic" coding style to languages like JavaScript or C. However, this elegant simplicity can be deceptive.

Research by Johnson et al [10] delves into the difficulties novices face with Python's operator overloading. The + operator, for example, can add integers or floats, concatenate strings, and extend lists, adding layers of complexity that may not be evident at first glance. Moreover, many relatively complex concepts lurk within Python's depths, potentially baffling beginners. These include meta-programming with decorators, higher-order functions (passing functions as arguments or return values), lambdas, closures, list and dictionary comprehensions, generators, coroutines, default parameters, ternary operators, type hints, and object-oriented features.

Even some essential Python features pose challenges to beginners. One such example is the indentation-based syntax. Whitespace usage often proves tricky for absolute novices: "How many spaces should I use? Is this a tab or a space? Does it even matter?" Unlike Python, most programming languages like JavaScript, Java, and C are whitespace-insensitive, meaning whitespace is purely for readability and handled automatically by code formatters. In Python, however, indentation dictates the program's structure, making incorrect usage akin to a syntax error.

Further compounding the challenges for novices is Python's dynamic type system. While seemingly straightforward, this feature harbors hidden complexities. For example, subtracting variables with numeric values is intuitive, but attempting the same with a string, even one convertible to a number, can lead to cryptic errors. Adding to the confusion, a variable's type can dynamically change depending on the program's execution path, introducing subtle, hard-to-trace bugs for inexperienced users.

Interestingly, both whitespace sensitivity and dynamic typing contribute to Python's allure of simplicity and elegance. They eliminate the need for explicit block closures with "end" statements or cumbersome curly braces, as well as the burden of constant type declarations. This perceived ease, however, comes at the cost of potential pitfalls for novices grappling with these implicit complexities. It's important to note that similar arguments can be made regarding the initial hurdles presented by other dominant novice languages like JavaScript, C, or Java, each with their own idiosyncrasies.

Block-based vs text-based languages

In a study comparing two introductory programming interfaces, Weintrop et al [25] tracked two matched classes at the same school. Both embarked on a 5-week curriculum, with one group using a block-based interface and the other a text-based version. Comprehensive assessments revealed that while both groups improved, the block-based interface fostered greater learning gains and deeper interest in future computer science endeavors. Interestingly, the text-based group viewed their experience as more professional and effective for skill development. This suggests that block-based interfaces may be ideal for initial learning and igniting interest, while text-based tools cultivate advanced skills and a professional perspective.

Weintrop et al [24] further highlight potential drawbacks to block-based environments. Students immersed in these frameworks may perceive them as less powerful and lacking in features compared to their text-based counterparts. This perception of "inauthenticity" could stem from a sense of not engaging with the deeper intricacies of "real", text-based code. Additionally, as project complexity grows, the inherent drag-and-drop interface can become increasingly cumbersome, leading to cluttered codebases that are more difficult to read and understand than their text-based equivalents. This negates the initial readability advantage block-based formats offer for simple programs to absolute novices.

While the engaging nature of block-based coding makes it an attractive option, effectively conveying foundational data structures and algorithms, like linked lists, binary trees, searching, sorting, and the aforementioned prime number algorithm, presents a significant challenge within a block-based framework. This necessitates the transition to text-based coding for a deeper understanding of these concepts.

Transitioning

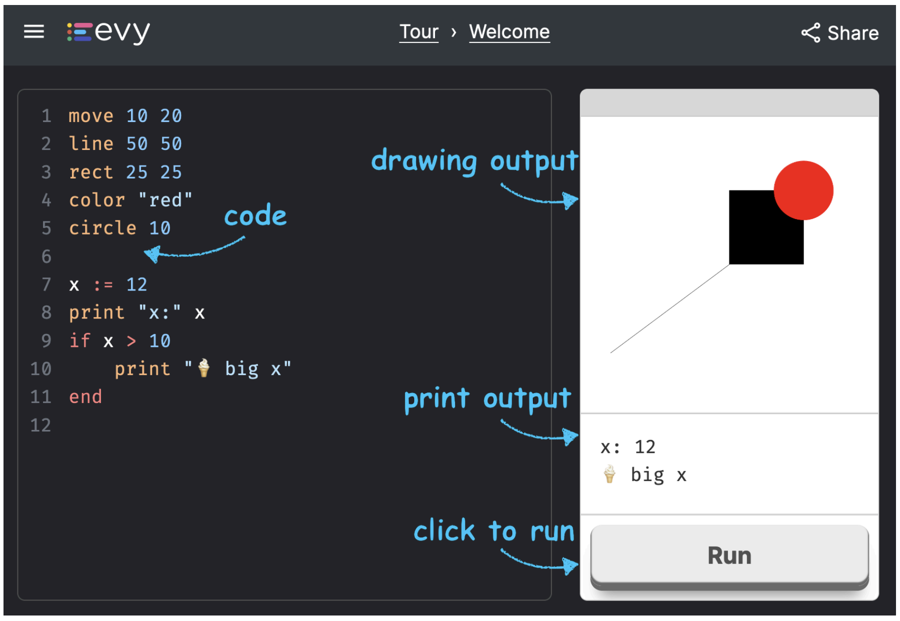
The dominant strategy for bridging the gap between block-based and text-based coding leverages preview modes. These, as Kölling et al [11] note, can take the form of software tools or readily accessible reference sheets, offering learners a familiar block-based visual representation alongside their corresponding textual code. Additionally, more recent hybrid systems, exemplified by Microsoft's MakeCode, allow for dynamic switching between code and block views [12]. However, this fluidity stumbles when encountering text-based features lacking a corresponding block equivalent.

Regardless of specific systems, certain core challenges confront learners transitioning from block-based to text-based programming [11, 27]:

- **Remembering Commands:** Block-based systems function as a readily accessible repository of commands, presented in a visually intuitive catalog. This approach empowers novices to explore the available functionality, refresh fading knowledge, and even spark innovative ideas by discovering new possibilities. In addition, text-based systems offer a wider range of functions than blocks exist in block-based environments, requiring familiarization with a larger command set.
- **Readability:** Readability remains a hurdle, as novices often find features like visually intuitive scope representation through bracketed blocks easier to grasp than textual equivalents with curly braces.
- **Precise Syntax:** Mastering text-based systems demands two levels of recall: the keyword and its exact use, including punctuation. Knowing a "for-loop" exists (and why) isn't enough; you need the keyword, comma placement, semicolons, and bracket dance all memorized. This layer of syntactic complexity adds another hurdle for new programmers, beyond simply remembering the commands themselves.
- **Typing:** Text entry via keyboard poses a significant challenge to the untrained novice, especially kids who have little to no experience with touch typing
- **Data Types:** While many block-based systems use a limited set of built-in types, allowing successful program creation without deep data type understanding, most text-based languages offer greater flexibility. They typically allow for custom types and specialized types like "datetime" requiring more nuanced comprehension. Even with dynamically typed languages, where types are implicit, users benefit from some data type familiarity to avoid potential errors.
- **Whitespace:** Indentation and spacing are done automatically in block-based systems by the shape and nesting of blocks. In text-based programming languages, it is often done manually. This is especially problematic for languages where whitespace is part of the syntax, such as Python.

D. Evy: A Simple, Engaging Text-based Programming Language

Evy is a simple procedural language with a limited static type system, bridging the gap for both students new to text-based programming and those transitioning from block-based environments. It retains the ease-of-use and engagement of block-based coding, letting students build and share animations and games online with zero setup (see Figure 4).


```
move 10 20
line 50 50
rect 25 25
color "red"
circle 10

x := 12
print "x:" x
if x > 10
    print "🍦 big x"
end
```

```
x: 12
🍦 big x
```

Figure 4: *Evy's web-based playground at https://play.evy.dev*

But Evy's power goes beyond fun: it equips students with enough syntax to learn foundational algorithms and data structures, laying the groundwork for success in CS and Software Engineering degrees. Compatible with diverse editors and environments, Evy adapts to your learning preferences as you advance.

Evy's syntax is purposefully minimalist, addressing the fact that especially young novice students do not yet have established touch typing skills and grasp shorter terms more easily. Whitespace in Evy is optional and only serves readability, but auto-indentation via code-formatter is built into Evy's web-based playground.

A hello-world program in Evy is as simple as:

```
print "Hello, world!"
```

Evy's static type system does not hide types, but makes it easy to declare typed variables through type inference. `n := 5` for instance declares the variable `n` of type number. There are only three basic types in Evy: `num`, `string`, `bool`.

There are two composite types: arrays `[]` and maps `{}`, and one dynamic type called `any`. Evy does not allow for custom type declarations via structs or classes. It completely lacks any OO and functional programming features and only allows for top-level function definitions. It caters to user interaction by allowing the definition of a predefined set of event handlers, with the `on` keyword, for example:

```
on down x:num y:num
    print x y
end
```

The code above prints the x and y coordinates on every pointer down event, mouse or touch.

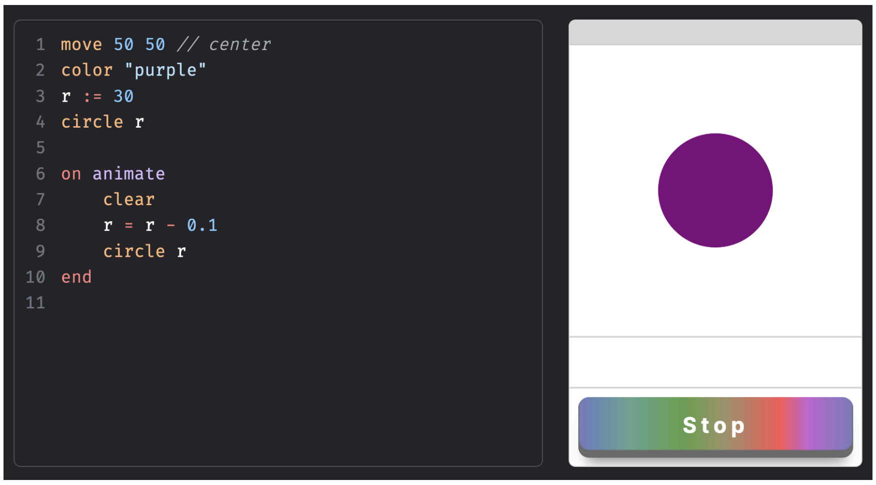
`if` statements have optional `else if` and `else` branches, but there is no "case" statement. There is a single continuation-condition `while` loop and a `for` loop that ranges over either arrays, maps or number ranges.

Evy has a small yet powerful set of builtin functions:

- **Input and Output**: print, read, cls, printf
- **Types**: len, typeof
- **Map**: has, del
- **Program control**: sleep, exit, panic
- **Conversion**: str2num, str2bool
- **Errors**: panic, err
- **String**: sprint, sprintf, join, split, upper, lower, index, startswith, endswith, trim, replace
- **Random**: rand, rand1
- **Math**: min, max, floor, ceil, round, pow, log, sqrt, sin, cos, atan2
- **Graphics**: move, line, rect, circle, color, width, clear, grid, gridn, poly, ellipse, stroke, fill, dash, linecap, text, font
- **Event Handlers**: key, down, up, move, animate, input

Evy outputs drawing commands to an area in the top right corner of the screen whose positions are defined by a Cartesian coordinate system, ranging from (0, 0) at the bottom-left to (100, 100) at the top-right. Traditional computer graphics often have an inverted y-axis with (0, 0) at the top-left. In contrast, Evy's coordinate system aligns with math classes, placing (0, 0) at the bottom-left. This consistency minimizes confusion for younger learners. For further details, see the Evy language specification, built-in documentation, and syntax by example (https://docs.evy.dev).

Figure 5 shows the Evy source code and a screenshot of a simple animation of a shrinking purple dot. Initially, we set the drawing pen color to purple and draw a circle with radius 30 at the center of the canvas (50,50). Then, we use the "on animate" event handler to update every frame, drawing a slightly smaller circle after clearing the canvas.



```
move 50 50 // center
color "purple"
r := 30
circle r

on animate
    clear
    r = r - 0.1
    circle r
end

```

Figure 5: *A Simple Animation in Evy - The Shrinking Purple Dot.*

Limitations

While Evy tackles many common hurdles for novice programmers, it still shares some inherent challenges. Syntax errors can occur, and case sensitivity and special characters demand precision and focus during text input. Evy also has room for improvement in readability of error messages and performance. Its creators are actively working on refining these aspects. Future plans include enhancing error messages to be clear and actionable, as well as optimizing execution performance for even smoother user experiences.

IV. Conclusion

Novice programmers routinely struggle to learn programming. Block-based environments, while easy-to-use, lack the power and authenticity needed for industry experience or foundational algorithms. Conversely, conventional text-based languages daunt many, especially younger beginners, with cryptic special characters and advanced concepts. Further research should be conducted on easing the transition from block-based to text-based coding and lowering the barrier to entry for learning text-based languages.

Evy exemplifies a text-based approach that bridges the gap between block-based and text-based methodologies, drawing on the strengths of both:

- **Minimizing cognitive load:** Evy embraces a minimalist syntax with fewer special characters and advanced concepts than most programming languages. This reduced complexity paves the way for faster grasp and internalization of its syntactic mechanics, thereby easing novices' path to algorithmic thinking.
- **Clarity and memorability**: Evy uses natural language keywords and has a small set of built-in functions that are easy to understand and remember, yet powerful enough for user interaction, games, and animations. These features empower engaging experiences, which in turn encourage efficient learning.
- **Focusing on Engagement:** Evy prioritizes sparking immediate excitement for beginners. With visually appealing graphics, interactive animations, and even game creation, it fuels motivation and makes learning fun through instant gratification. This goal is further supported by Evy's zero-setup approach: go to the website, look at samples, and code away!
- **Learning from the past and present:** Evy draws inspiration from both the accessibility of legacy languages like Basic and Pascal and the best practices of modern languages. Its minimalist syntax reflects past languages, while features like automated formatting integrate modern advancements. Additionally, Evy's online playground and example-rich documentation mirror the readily available resources found in contemporary platforms.
- **Seamless transition to advanced languages:** With a firm grasp of fundamentals in Evy, like algorithmic thinking and core data structures, transitioning to Python, Javascript or other modern conventional programming language becomes considerably easier compared to starting from scratch or solely relying on block-based coding.
- **Making sharing easy:** Evy fosters a sense of community among young programmers by enabling instant sharing of creations via unique URLs. This promotes collaboration, feedback, and a sense of accomplishment, making the learning experience even more rewarding.

While the Evy language itself is complete, our ambition extends beyond code. We're actively developing a research-backed learning platform and course framework to address shortcomings of text-based programming education, eliminating the barriers that often deter beginners. To achieve this vision, we seek a supportive home for the Evy project.

Ideally, we'd partner with a university or research lab with expertise in educational technology. However, we welcome all forms of support, whether you're an educator passionate about outreach, a developer interested in contributing to our open-source platform, or a sponsor eager to invest in the future of programming education.

References

[1] Altadmri, A., & Brown, N. C. (2015, February). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education* (pp. 522-527).

[2] Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to “real” programming. *ACM Transactions on Computing Education (TOCE)*, *14*(4), 1-15.

[3] Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: blocks and beyond. *Communications of the ACM*, *60*(6), 72-80.

[4] Ball, T., Chatra, A., de Halleux, P., Hodges, S., Moskal, M., & Russell, J. (2019, October). Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (pp. 7-12).

[5] Cheah, C. S. (2020). Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology*, *12*(2), ep272.

[6] Cooper, S. (2010). The design of Alice. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1-16.

[7] Chow, S., Yacef, K., Koprinska, I., & Curran, J. (2017, July). Automated data-driven hints for computer programming students. In *Adjunct publication of the 25th conference on user modeling, adaptation and personalization* (pp. 5-10).

[8] Grover, S., & Basu, S. (2017, March). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education* (pp. 267-272).

[9] Harvey, B. (2019). Why do we have to learn this baby language?.

[10] Johnson, F., McQuistin, S., & O'Donnell, J. (2020, January). Analysis of student misconceptions using Python as an introductory programming language. In *Proceedings of the 4th Conference on Computing Education Practice* (pp. 1-4).

[11] Kölling, M., Brown, N. C., & Altadmri, A. (2015, November). *Frame-based editing: Easing the transition from blocks to text-based programming*. In Proceedings of the Workshop in Primary and Secondary Computing Education (pp. 29-38).

[12] Lin, Y., & Weintrop, D. (2021). The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages*, *67*, 101075.

[13] Luxton-Reilly, A. (2016, July). Learning to program is easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 284-289).

[14] Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1-15.

[15] Miskin, H., & Gopalan, A. (2017). Skramble: An embeddable python programming environment for use in learning systems. In *Computers Supported Education: 8th International Conference, CSEDU 2016, Rome, Italy, April 21-23, 2016, Revised Selected Papers 8* (pp. 193-213). Springer International Publishing.

[16] Papert, S. A. (2020). *Mindstorms: Children, computers, and powerful ideas*. Basic books.

[17] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137-172.

[18] Sobral, S. R. (2019). 30 Years of CS1: Programming languages evolution.

[19] Sun, D., Looi, C. K., Li, Y., Zhu, C., Zhu, C., & Cheng, M. (2024). Block-based versus text-based programming: a comparison of learners' programming behaviors, computational thinking skills and attitudes toward programming. *Educational technology research and development*, 1-23.

[20] Tempel, M. (2013). Blocks programming. *CsTA Voice*, *9*(1), 3-4.

[21] Trower, J., & Gray, J. (2015, February). Creating new languages in Blockly: Two case studies in media computation and robotics. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 677-677).

[22] Utting, I., Cooper, S., Kölling, M., Maloney, J., & Resnick, M. (2010). Alice, greenfoot, and scratch--a discussion. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1-11.

[23] Watson, C., & Li, F. W. (2014, June). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 39-44).

[24] Weintrop, D., & Wilensky, U. (2015, June). To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children* (pp. 199-208).

[25] Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1-25.

[26] Weintrop, D., & Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education*, *142*, 103646.

[27] Weintrop, D. (2021). The role of block-based programming in computer science education. *Understanding computing education*, *1*, 71-78.

[28] Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, *51*(1), 71-87.

[29] Wirth, N. (2000). The Development of Procedural Programming Languages: Personal Contributions and Perspectives. *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*. Springer-Verlag, Berlin, Heidelberg, 1–10.

[30] Yücel, Y., & Rızvanoğlu, K. (2019). Battling gender stereotypes: A user study of a code-learning game,"Code Combat," with middle school children. *Computers in Human Behavior*, *99*, 352-365.

Julia Ogris has over 20 years of industry experience building software at innovative startups and established corporations including Google and ANZ Bank. She noticed a gap in beginner-friendly programming tools. Passionate about inspiring the next generation to code, she collaborated with her young daughters to create Evy, a bridge between intuitive block-based environments and powerful real-world languages (https://evy.dev).