Enhancing Block Composition: The Role of Category Highlighting in Block-Based Environments

Mauricio Verano Merino
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
m.verano.merino@vu.nl

Niels Kok
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
nielskokrl@gmail.com

***Abstract*—Block-based programming environments have become widely recognized as a user-friendly approach to programming, especially for beginners and non-technical users. They offer a programming experience based on the what-you-see-is-what-you-get (WYSIWYG) paradigm. These environments employ visual jigsaw-like blocks that users can snap together to form programs, allowing them to focus on logical concepts without the burden of the language's syntax. However, the usability of these environments often varies, affecting the quality of the user experience. This paper presents *Category Highlighting*, a technique designed to enhance block discoverability, facilitate intuitive interactions, and improve overall navigation within environments built on the Google Blockly library. We demonstrate its usefulness in different case studies.**

***Index Terms*—Usability, Syntax, Grammars, Kogi, Block-based Environments, Blockly, Visual Programming**

I. Introduction

Over the past few years, block-based programming languages have gained widespread popularity, emerging as an ideal entry point for beginners eager to explore the world of programming [1]–[3]. These languages utilize visual blocks that can be easily snapped together to create programs, providing an intuitive and engaging learning experience.

Block-based environments are visual programming platforms that represent language constructs using interlocking, jigsaw-like blocks. Each construct is depicted with distinct block shapes and edge features that provide visual hints about how the blocks can be connected. The advantage of this type of interface is that it offers a "what-you-see-is-what-you-get" (WYSIWYG) programming experience while eliminating the possibility of syntax errors [4]–[7]. A block-based editor functions as a tool for directly manipulating the abstract syntax of a language, making block-based programming a form of projectional editing.

Among the tools that facilitate the development of block-based environments are Kogi [8] and its enhanced version, S/Kogi[1] [9], [10]. Kogi simplifies the creation of visual block-based programming environments by generating them directly from language specifications. It leverages Google Blockly [11], a widely used library for building block-based programming interfaces, to provide an intuitive and adaptable development framework. Although Kogi-like tools show great promise in generating user-friendly block-based environments, the introduction of new enhancements can further improve their usability. To address this, this paper focuses on the following research question. *How can block composition be made more explicit in block-based environments?*

The remainder of this paper is structured as follows: We first provide background information on block-based environments (Section II) and introduce Kogi (Section III). We then explore the challenge of an unclear block composition (Section IV) and propose a solution, *Category Highlighting* (Section V). To examine its impact, we present findings from four case studies (Section VI). Finally, we discuss the implications of introducing *Category Highlighting* (Section VII) and conclude with future research directions (Section VIII).

II. Block-based Environments

A block-based environment is a visual, interactive drag-and-drop programming interface where language constructs are depicted as jigsaw-like puzzle pieces, referred to as blocks. These blocks are designed with distinct visual features, such as shape, color, and connection patterns. These features assist users in grasping how various blocks (language constructs) can be combined to form valid programs [12]. For example, Figure 1 shows a block-based representation of a `while` loop. The red circle in Figure 1 indicates a user input. Inputs are blocks that users can drag into other blocks to form a syntactically valid program, also known as *Fields*. *Fields* represent data inputs, with common types including numbers, strings, lists, and variables.

In addition to *Field* inputs, there are *connections* that guide users through visual cues, such as male and female jigsaw connections, showing that blocks can be combined. The green circle (Figure 1) illustrates a male jigsaw connection, indicating a required Statement input. These inputs allow blocks to be vertically stacked, building a coherent program.

Toolbox

The Google Blockly toolbox is a key component, providing a palette of blocks for users to build programs. Organized into color-coded categories like loops, events, and variables, it enables users to easily locate and drag blocks into the

[1]S/Kogi builds upon Kogi by incorporating a set of heuristics designed to improve the usability of the generated environments.

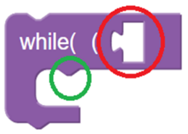


Fig. 1: A while block created in Google Blockly [12].

workspace to construct sequences of instructions. This organization enhances user experience by promoting a systematic approach to programming and helping users quickly find the right blocks for each task.

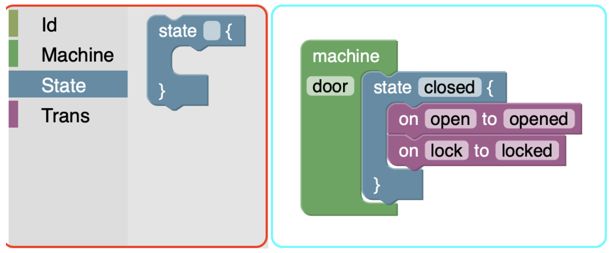


Fig. 2: A typical Block-based environment using the State Machine language (Listing 1).

The red area in Figure 2 is the environment's toolbox. In this example, the category 'Image' has been selected and the corresponding blocks can be viewed by the user.

Canvas

The Blockly canvas (blue highlighted area in Figure 2) is the core area in Block-Based environments, it is where users build programs with visual blocks. It allows blocks to be dragged, connected, and arranged in sequences. This intuitive interface helps beginners to understand programming concepts.

III. Kogi

Kogi is a language-parametric block-based generator that derives block-based environments from language specifications [8], [13]. It is developed using Rascal [14] for language definition and Google Blockly for visualization of the resulting environments. Kogi traverses the context-free grammar definition of the languages to extract essential syntactical information to derive blocks that align with the language's specification.

Listing 1 presents an example of a grammar for a state machine language. The grammar contains four labeled production rules (e.g., state, and transition) that define the language's syntax. Within these productions, a combination of nonterminal (e.g., `State`, and `Trans`) and terminal symbols (e.g., `"on"` and `"to"`) play a crucial role, as we will use them to improve Kogi's generated environments. Each nonterminal symbol can be labeled (e.g., `id` in the nonterminal symbol `Id`). These labels are essential in understanding Kogi's block derivation process and will help us explore the limitations and potential enhancement to the Blockly library.

```
start syntax Machine =
   machine: "machine" Id id State* states;
syntax State=
   state: "state" Id id "{" Trans* transitions "}";
syntax Trans =
   transition: "on" Id "to" Id to;
lexical Id = id: [a-zA-Z]+;
```

Listing 1: State machine grammar using Rascal.

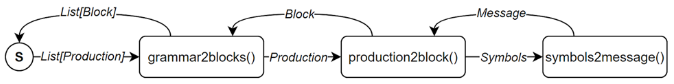


Fig. 3: The path from a list of Productions to a list of Blocks.

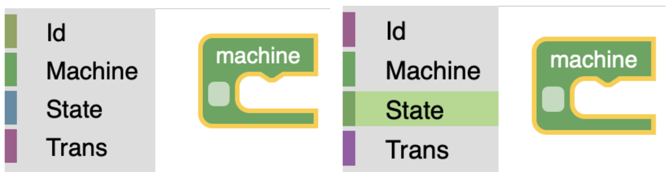
From Production Rules to Blocks: To gain a comprehensive understanding of the inner workings of Kogi, it is crucial to delve into the step-by-step process it employs to derive a Blockly environment from a context-free grammar. Now, let's examine how Kogi transforms production rules into blocks.

Figure 3 presents a simplified overview of the key steps involved in this process. It begins on the left with a context-free grammar. Leveraging the capabilities of the Rascal metaprogramming language [15], we extract a list of production rules from the grammar. This list is then provided as input to the `grammar2blocks` module, which transforms the production rules into blocks. Notably, the transformation process is not confined to the `grammar2blocks` module alone; instead, it depends on auxiliary modules, such as `symbols2message`, to facilitate the conversion of production rules into blocks. This module, the `symbols2message` module, takes the terminal and nonterminal symbols of each production rule as input and converts these symbols into messages, which is the mechanism offered by Blockly to enable developers to define custom block configurations. Once `grammar2blocks` generates a list of blocks, the block generation process remains incomplete. The final step involves creating the *Toolbox* component, which organizes the generated blocks into categories.

The Toolbox generation process operates as follows: each nonterminal symbol N is mapped to a corresponding category. For every nonterminal symbol, a category named N is created. Subsequently, all production rules (P) associated with each nonterminal symbol N are assigned to their respective category N, and the generated blocks are named P. The type name of each block, which serves as an identifier within the block-based environment (invisible in the user interface), is set to N/P. The remainder of the block's data structure is then populated with precise details based on its terminal and nonterminal symbols. Finally, the blocks and the Toolbox are compiled into JavaScript and HTML for use in the environment.

IV. Challenges in Block Composition and User Guidance in Block-Based Environments

With a clear understanding of context-free grammars, the Blockly environment, and Kogi, we now address an ongoing



(a) Limitation. (b) Solution.

Fig. 4: Example of the category highlighting limitation (left) and proposed solution (right).

issue in *S/Kogi* (Kogi's improved version) [9]. A noteworthy challenge that emerged when working with Kogi-generated environments—and, more broadly, with Blockly-based environments—is the lack of clear instructions for users on how to connect blocks effectively. The absence of labeled inputs on the blocks leaves users uncertain about the appropriate category and specific block to use when connecting the inputs of a block. For example, for the State Machine language (Listing 1), Kogi will generate four categories (one for each labeled production) as shown in Figure 2. However, there is no guidance indicating which category should be used for the open inputs of a block once it is placed in the workspace.

Although this issue is relatively manageable in a small language like the State Machine language, it becomes significantly more pronounced in realistic languages with a larger number of constructs (e.g., MiniJava). As a result, users face limited visibility and reduced role-expressiveness, often relying on guesswork to identify the correct block type. This is particularly challenging for beginners and novice users of Kogi and block-based environments in general, as it demands an understanding of the underlying grammar structure to navigate the block-based environment effectively. Such a requirement contradicts the purpose of these environments, which aim to simplify programming for end-users.

The lack of visual cues or clear communication regarding which blocks can connect to others could hinder users' experience when programming. For instance, in the state machine example (Listing 1), when a user creates an empty state machine on the canvas (see Figure 4a), no visual indicators specify the valid blocks (language constructs) that can connect to the current program. This gap in guidance leaves users to deduce compatibility on their own. Notably, all the necessary information to assist users in program development already exists within the block-based environment or the underlying grammar, yet it remains inaccessible or underutilized in practice.

V. Category Highlighting to Enhance Block Composition in Blockly

To address the issue of unclear block composition (Section IV), we developed *Category Highlighting* as a solution for block-based environments. Implemented as an extension to Kogi, this feature enhances user guidance by visually highlighting the relevant categories for all inputs of a selected block. This highlighting feature provides a clear visual cue, helping users identify compatible blocks to connect. It does so by coloring the background of the categories with a light green color, enhancing the overall visibility when trying to compose blocks in the workspace. If an input has already been occupied by a block of the same category, the background turns orange, further improving clarity and streamlining the block composition process.

To implement *Category Highlighting*, we leverage the event listener feature provided by Blockly. Event listeners are functions that continuously monitor user actions within the Blockly workspace. Each action is associated with a unique event name. For instance, clicking on a block triggers an event. By utilizing the event listener, we can extract the ID of the selected block, enabling us to leverage the built-in functions of the Blockly library and apply them to this block. One of these functions allows us to extract all information related to the inputs of a block, including which type of blocks are allowed and whether the input is already populated. Remember that the type of block tells us in which category the block is stored. Now, we can simply cross-reference the allowed types with the categories available in the toolbox, and if the type matches with a category, we store the category to be colored later. If the input is already occupied, we give it an orange color; otherwise, we color it green. This is how we indicate to a user which categories are associated with the inputs of a block.

An example is shown in Figure 4b, where a block from the `Machine` category has been dragged into the workspace and selected. With *Category Highlighting* enabled, the relevant categories associated with the block's inputs are visually highlighted. From these highlights, we can see that the selected block requires inputs from the `State` category, providing clear guidance for block composition.

This feature reduces the cognitive burden on users by alleviating the need to memorize the underlying grammar structures of the languages. With *Category Highlighting*, users receive instant visual feedback, reducing reliance on explicit grammar knowledge and streamlining the block selection process. By providing clear indications of block compatibility, this feature facilitates informed decision-making, supports systematic exploration within the Blockly environment, and enhances overall usability. Ultimately, by minimizing cognitive load, we believe that, *Category Highlighting* contributes to a more efficient and user-friendly block-based programming experience.

VI. Case Studies

Using Kogi, we generated block-based environments for four distinct languages: the state machine language (discussed earlier), along with CCL, Pico, and QL — all implemented using Rascal. Below, we provide an overview of the latter three emphasizing the impact of supporting *Category Highlighting* across different languages. This enhancement aims to improve the user experience by reducing cognitive load and streamlining the block-building process within Blockly's canvas.

A. CCL

The Cloud Configuration Language (CCL) is a specialized language modeled after the structure used in Amazon Web Services CloudFormation [16] for provisioning cloud resources. Unlike traditional domain-specific languages (DSLs) or general-purpose languages (GPLs), CCL features a largely fixed structure, where the primary variations occur in the values provided.

Figure 5 illustrates an example program written in the *CCL* language, where a user aims to allocate cloud resources for an application. To achieve this, the user employs a `Resources` block and adds an instance. In the figure, some parameters for the `instance` have already been defined; the corresponding categories are highlighted in orange. This differentiation is significant because it allows users to concentrate on the categories highlighted in green, which provide guidance and assistance for completing the program.

Building on this, Figure 6 demonstrates a different scenario where the user adds a new instance without any predefined parameters. In this case, all parameters are initially empty, and the valid categories are highlighted in green (e.g., CPU, Image, IPV6), guiding the user through the parameter selection and configuration process.

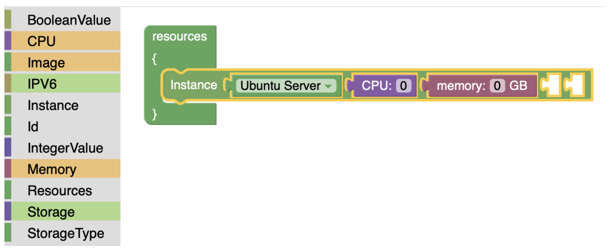


Fig. 5: Example CCL program to define a cloud instance with some prefilled parameters.

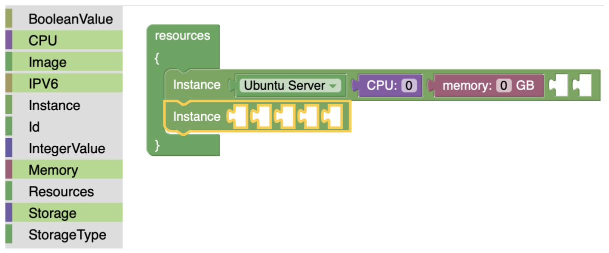


Fig. 6: Example CCL program to define a cloud resource without prefilled parameters.

B. Pico

Pico is a simplified programming language, similar to the While language, commonly used in textbooks on programming language semantics. An implementation of Pico is included in the Rascal standard library [17]. To develop a block-based interface for Pico, we leveraged its existing grammar from the Rascal library as input to Kogi. The resulting block-based environment is illustrated in Figures 7 and 8, which already supports *Category Higlighting*.

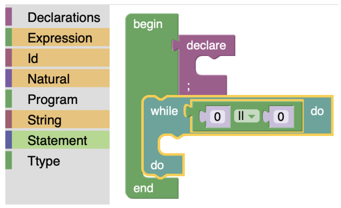


Fig. 7: Example Pico program to define a simple while loop.

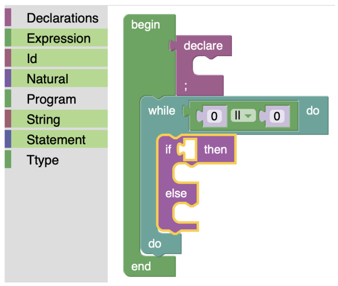


Fig. 8: Example Pico program to define a simple while loop.

C. QL

QL is a domain-specific language (DSL) designed for creating interactive questionnaires and has been used as a benchmark for evaluating language workbenches [15]. Its suitability for block-based environments stems from its focus on non-programming tasks, making it particularly accessible to domain experts who may have little to no programming experience. Block-based interfaces, with their intuitive features such as natural language labels, distinct colors and shapes, and drag-and-drop interactions, can provide a more user-friendly experience for this audience [50, 52].

Since Rascal already included an implementation of QL, we utilized its existing syntax to generate a block-based version. Specifically, QL's concrete syntax in Rascal was used as input for Kogi to produce the block-based environment, including the category highlighting feature. Figure 9 illustrates an example of a tax questionnaire created within this environment, where a domain expert could design a simplified tax form with a basic question (`hasSoldHouse`). In this example, we can observe an incomplete conditional inside the `hasSoldHouse` question. Thanks for the category highlighting; the environment is highlighting to the user that they must use a block from the `Expr` or `Id` categories to complete the definition of the conditional block.

VII. Discussion

The introduction of *Category Highlighting*, aimed at improving visual clarity and user experience within Blockly environments. To the best of our knowledge, this is the first highlighting system for block-based environments. While

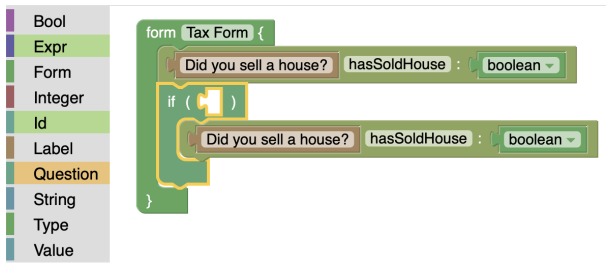


Fig. 9: Example QL program to define a simple tax form.

the impact of this improvement on user interaction and engagement is evident in qualitative observations, and personal experience, there is the need of conducting a user study to understand the impact of this feature in practice.

We observed and experienced that the inclusion of this feature enhances the visual representation of the Toolbox categories, making it easier for users to distinguish between different block categories and fostering a more intuitive programming experience. However, due to time constraints and limited access to a diverse user base, gathering user feedback was not feasible. Nevertheless, we tried the features in four different languages as shown in Section VI, and informally, users inside the authors' network who tried the *Category Highlighting* feature expressed positive responses, citing improved ease of use and reduced cognitive load when navigating through the block-based environment canvas. Their feedback and our experience suggest that the enhancement would likely contribute to the overall improvement of the Kogi-generated environments, and Blockly environments in general.

VIII. Conclusions & Future Work

The case studies in Section VI suggest that adding *Category Highlighting* in visual programming environments may enhance the user experience within Blockly environments. However, it is essential to acknowledge certain limitations and considerations that emerged during the study. Specifically, the case study on the State Machine language highlighted that certain languages may not experience significant improvements due to their inherent simplicity. Identifying the types of languages most likely to benefit from these enhancements would be valuable for tailoring improvements to better suit specific use cases.

In conclusion, this paper presents improvements in Block-based environments implemented within Kogi that might enhance user experience in visual block-based environments. To ensure further progress, it is crucial to consider the challenges and nuances that may arise. By prioritizing user needs and finding the right balance between block reduction and code readability, we can create a more effective and refined Kogi, enhancing the user experience to an even greater extent.

Overall, this exploration of improving block-based environments lays the groundwork for further research and development within the context of improving user experience in Blockly environments using Kogi. Some suggestions for future work and investigation:

User Studies: Conducting large-scale user study with diverse participants will provide a more in-depth understanding of the effectiveness of the proposed improvement. Gathering feedback from practitioners and novices with a diverse programming levels and tasks would yield valuable insights.

Language-Specific Enhancements: Tailoring Kogi's improvements to specific programming languages could maximize their impact. Investigate language-specific features that may benefit from optimization and streamline the code generation process accordingly.

User-Centric Design: Apply user-centric design methodologies to continuously improve the programming experience using visual programming environments based on feedback and user needs. This approach ensures that future iterations of Kogi are well-suited to the preferences and requirements of the specific-programming community.

Cross-Platform Support: Investigate ways to make Kogi compatible with other block-based programming environments and existing IDEs (e.g., VSCode or JetBrains IDEs).

References

[1] D. Weintrop, "Block-based programming in computer science education," *Communications of the ACM*, vol. 62, pp. 22–25, 07 2019.

[2] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming," *Communications of the ACM*, vol. 60, pp. 72–80, 05 2017. [Online]. Available: https://arxiv.org/pdf/1705.09413.pdf

[3] D. Weintrop and U. Wilensky, "To block or not to block, that is the question," *Proceedings of the 14th International Conference on Interaction Design and Children*, 06 2015.

[4] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 91–99.

[5] L. Moors and R. Sheehan, "Aiding the transition from novice to traditional programming environments," in *Proceedings of the 2017 Conference on Interaction Design and Children*, ser. IDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 509–514.

[6] D. Weintrop and U. Wilensky, "Between a block and a typeface: Designing and evaluating hybrid programming environments," in *Proceedings of the 2017 Conference on Interaction Design and Children*, ser. IDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–192.

[7] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating coblox: A comparative study of robotics programming environments for adult novices," *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pp. 1–12, 2018.

[8] M. Verano Merino and T. van der Storm, "Block-based syntax from context-free grammars," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–295. [Online]. Available: https://doi.org/10.1145/3426425.3426948

[9] *Getting grammars into shape for block-based editors*. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3486608.3486908

[10] T. Beckmann and M. Verano Merino, "S/kogi." [Online]. Available: https://doi.org/10.5281/zenodo.5534113

[11] E. Pasternak, R. Fenichel, and A. Marshall, "Tips for creating a block language with blockly," 2017. [Online]. Available: https://developers.google.com/blockly/publications/papers/TipsForCreatingABlockLanguage.pdf

[12] "Blockly — google developers," Google Developers, 2019. [Online]. Available: https://developers.google.com/blockly

[13] M. Verano Merino and T. van der Storm, “Kogi.” [Online]. Available: https://doi.org/10.5281/zenodo.4033220

[14] “The rascal meta programming language — the rascal meta programming language,” www.rascal-mpl.org. [Online]. Available: https://www.rascal-mpl.org/

[15] P. Klint, T. van der Storm, and J. J. Vinju, “Easy meta-programming with rascal. leveraging the extract-analyze-synthesize paradigm for meta-programming,” in *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE’09)*, ser. LNCS. Springer, 2010.

[16] A. W. Services. (2024) Aws cloudformation documen- tation. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-ec2-instance.html

[17] Rascal. (2017) Pico. [Online]. Available: https://www.rascal-mpl.org/docs/Recipes/Languages/Pico/