

# Block-Based Syntax from Context-Free Grammars

Mauricio Verano Merino  
m.verano.merino@tue.nl

Eindhoven University of Technology  
Eindhoven, The Netherlands

Tijs van der Storm  
storm@cw.nl

CWI - University of Groningen  
Amsterdam, The Netherlands

## Abstract

Block-based programming systems employ a jigsaw metaphor to write programs. They are popular in the domain of programming education (e.g., Scratch), but also used as a programming interface for end-users in other disciplines, such as arts, robotics, and configuration management. In particular, block-based environments promise a convenient interface for Domain-Specific Languages (DSLs) for domain experts who might lack a traditional programming education. However, building a block-based environment for a DSL from scratch requires significant effort.

This paper presents an approach to engineer block-based language interfaces by reusing existing language artifacts. We present Kogi, a tool for deriving block-based environments from context-free grammars. We identify and define the abstract structure for describing block-based environments. Kogi transforms a context-free grammar into this structure, which then generates a block-based environment based on Google Blockly. The approach is illustrated with four case studies, a DSL for state machines, Sonification Blocks (a DSL for sound synthesis), Pico (a simple programming language), and QL (a DSL for questionnaires). The results show that usable block-based environments can be derived from context-free grammars, and with an order of magnitude reduction in effort.

**CCS Concepts:** • Software and its engineering → Visual languages; Domain specific languages; Graphical user interface languages.

**Keywords:** block-based environments, DSLs, language workbenches, grammars, visual languages, syntax, Blockly, Rascal

## ACM Reference Format:

Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org). SLE '20, November 16–17, 2020, Virtual, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8176-5/20/11...\$15.00

<https://doi.org/10.1145/3426425.3426948>

ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3426425.3426948>

## 1 Introduction

Block-based environments have received much attention in recent years due to their ease of use for non-programmers [6]. Block-based environments are visual programming environments that use jigsaw-like blocks to represent language constructs. Each language construct is represented using different block-shapes with visual cues on the edges that indicate how blocks can be connected.

For instance, the following shows a possible block-based representation of an if-statement:



The hole next to the if-level indicates the shape of expressions that are allowed there, and the dent between the curly braces indicates which kind of blocks can be nested under the if-statement. The benefit of such an interface is a what-you-see-is-what-you-get (WYSIWYG) programmer experience and the impossibility of syntax-errors [34, 38, 50, 53]. A block-based editor essentially is an editor to manipulate the abstract syntax of a language. Thus, editing block-based programs can be seen as a form of projectional editing.

Block-based environments have seen many uses in the software engineering field [1, 2, 11, 37, 50, 51]. They have also been widely investigated as educational tools [9, 13, 22, 25, 31, 46, 49]. However, developing block-based environments currently lacks solid engineering principles, which leads to ad-hoc implementation using various technologies and frameworks. As a result, the block-based language definition is hidden in arbitrary, general-purpose programming code. Moreover, this hinders the reuse of existing language artifacts, such as type checkers, interpreters, and compilers.

One way to ease the development of block-based environments is with libraries such as Google Blockly [36], Droplet [5], or Open-Blocks [42]. Another way is to extend existing block-based environments like Scratch [41], MIT App Inventor [37], or Snap! [33]. Many applications have been developed using these two alternatives. For example, Zhou et al. [55] developed a block-based language for teaching Latin grammar using Blockly. Likewise, Breuch et al. created Airblock [8]

using Scratch to foster block-based programming and aerodynamics principles. However, although these libraries and tools help in the development process of block-based environments, these solutions are still based on copying and modifying existing low-level code.

In this paper, we present an approach, Kogi, to derive block-based languages from declarative context-free grammars, such as used in language workbenches like Rascal [29], Spoofox [26], and Xtext [16]. This opens the possibility to reuse existing grammars and language artifacts already developed using such language workbenches. Kogi is implemented in Rascal. It reflectively transforms Rascal's built-in context-free grammars into an abstract representation of block-based user-interfaces, which is then compiled to Google Blockly [18] code. As a result, both existing and new DSL implementations in Rascal can be provided with a block-based interface with minimal effort.

The contributions of this paper can be summarized as follows:

- We dissect the structure of block-based environments and model it using an abstract grammar (Section 2).
- We present Kogi, a tool that analyzes context-free grammars in Rascal and derives a block-based environment using Blockly's API (Section 3). The implementation of Kogi, along with documentation and examples, is available on Github<sup>1</sup>.
- We present how the simplification of a context-free grammar impacts the complexity of the generated block-based environment (Section 3).
- Kogi's utility is demonstrated by generating block-based interfaces for four languages: State machines, Sonification Blocks, Pico, and QL (Section 4).

We conclude this paper with a discussion of further directions (Section 5), related work (Section 6), and concluding remarks (Section 7).

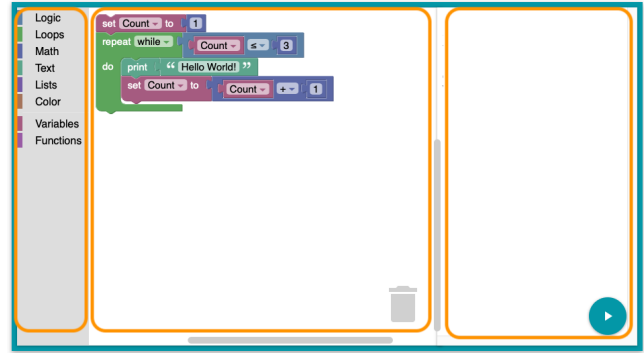
## 2 Anatomy of Blockly

This section describes what a block-based environment is and its parts.

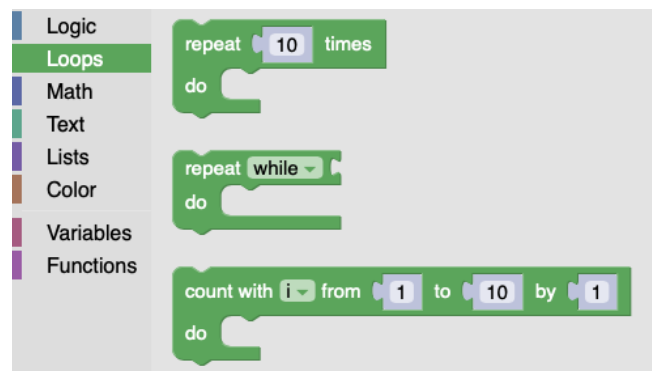
A block-based environment is a visual programming environment that uses blocks as language constructs. This paper focuses on block-based environments that adopt the jigsaw metaphor. One of the most known examples of this kind of environment is Scratch [41]. Scratch is a platform that uses a block-based environment for creating interactive stories, games, and animations. However, there are many more applications of a block-based environment to a diverse range of domains, including a wide range of domains such as aerodynamics [8], music [4], robotics [50], software engineering [1, 37], arts [45], and biology [17].

Looking at several block-based environments, we split them into three components: a *toolbox*, a *canvas*, and an

<sup>1</sup><https://github.com/cwi-swat/kogi>



**Figure 1.** A block-based environment built with Blockly [18].



**Figure 2.** Toolbox shelf [18].

*execution pane*. It is important to remark that the names of these components vary from one block-based environment to another. Figure 1 shows a typical example of a block-based environment built using Blockly. The following subsections explain each element in more detail, using Blockly derived environments as a representative style.

### 2.1 Toolbox

The *toolbox* in a block-based environment is a group of shelves (block categories) that contain all the language constructs of a block-based language (left view of Figure 1). Each language construct is represented as a block (as the if-statement shown in Section 1). A toolbox is often divided into several shelves, with a specific label, color, and group of language constructs (blocks). A shelf is used by developers to group language constructs according to some criteria. From the end-user perspective, how the toolbox shelves are organized is essential because it affords blocks' discoverability [23]. Figure 2 shows an example of how one of these shelves look like.

### 2.2 Canvas

The canvas (middle view of Figure 1) is where the user creates programs (scripts). Block-based programs are created by

dragging and snapping blocks together from the toolbox into the canvas [54]. The middle view of Figure 1 shows an example of a script created using a block-based language.

### 2.3 Execution View

The execution view (right view in Figure 1) is often placed next to the canvas view. This view is used mainly for two tasks, to interact with the current script in the canvas (e.g., execute); and display the script's execution output. The elements of this view vary quite a lot, depending on the language. For instance, as a result of computing scripts, some environments produce animations (e.g., Scratch), while some others do not display anything, but instead, they control external hardware (e.g., robots).

### 2.4 Blocks

The keystone of a block-based environment, as its name suggests, are jigsaw-like blocks. A block is the atom of a block-based language; it represents the language's syntax. Each block is a visual element that provides visual cues to the user about the meaning of the block, how it can be instantiated, and where it can be placed to create meaningful block-based programs. Each block is different from another in a block-based language, yet they have four typical elements: shape, label, color, and connections.

Following Blockly's approach, a block is defined by five elements, namely, their *inputs*, *fields*, *connections*, *colors*, and *tooltips*.

**2.4.1 Block Input.** In a block-based environment, the block's input is the information required to define a block, meaning it represents other blocks' possible connections. A single block might have one or more inputs, and each input is represented with labels and fields that shows its possible connections [19]. There are mainly three different types of input, namely, *value*, *statement*, and *dummy*. The type of input denotes the shape of each block. Furthermore, a block-based environment allows developers to define how they want to show the input fields; there are two types, *external* (condition of the *if* block in Figure 3) inputs and *inline* (condition of the *if* block in Figure 4).



Figure 3. If block with an external input value.

**Value input.** This element is used to stack blocks horizontally. Thus, it is frequent to use them for defining expressions. *Value inputs* are connected to the output connection

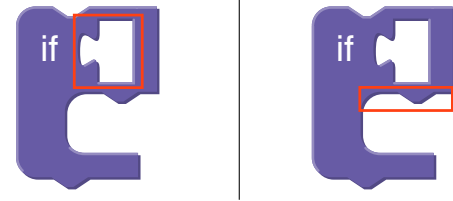


Figure 4. If block that highlights the usage of an input value block (left) as its first argument (condition of the *if* statement) and a statement input (right) as its second argument (body of the *if* statement).

of a value block. For instance, an *if* block condition (Figure 4) is represented using a value input, allowing Boolean expression blocks to be snapped horizontally.

**Statement input.** It is used to stack blocks vertically. As its name suggests, this type of block is used to represent statements. For example, the body of an *if-else* statement is represented with a statement input. Likewise, the block-based representation of the *if* statement (Figure 4 - right) uses a statement input. The red square denotes where the following blocks can be snapped in.

**Dummy input.** It is mostly used for adding layout to the blocks (e.g., adding labels or new lines). It does not create or allow new block connections.

**2.4.2 Fields.** Fields are used within blocks to represent input (literal) data from the user. There are different types of fields, depending on their data types, and each of them has different visual cues that help end-users fill in the right information. Some of the most common field types are *string*, *numbers*, *images*, *dropdown lists*, *checkboxes*, *colors*, and *variables*. However, block-based platforms allow developers to create their custom fields.

**2.4.3 Connections.** The block's connections offer a visual cue to guide end-users to compose blocks to create meaningful applications. Each block-based environment might have slightly different ways of representing connections. In this paper, we will illustrate this using Blockly's UI. There are three types of block connections: *no connection*, *left output*, and *top & bottom connection*.

**No connection.** This connection means that the block cannot be stacked to other blocks, yet this does not mean that it cannot contain other blocks. An example of this type of connection is shown in the *if* block in Section 1.

**Left output.** This connection is visually represented as a male jigsaw connector [20]. Blocks with a left output are often used to create values, and they are connected to *value inputs*. Blocks that produce an output cannot have a *previous* nor *next* statement connection.

**Previous-next connection.** There are three different ways of using this connection. Developers can define the block either with a *previous* connection, *next* connection, or both. The *previous* connection in a block is represented with a notch on its upper part. This notch enables it to be connected to a stack of blocks. Moreover, the *next* connection is represented with a bump at the block's bottom to allow other statement blocks to be stacked below it. Finally, blocks that support both *previous* and *next* connections are represented with both a notch and a bump in the upper and bottom parts, respectively. Figure 4 presents an example of a block that supports *previous* and *next* connections.

## 2.5 The Grammar of Blockly

In the above subsection, we have described the high-level structure of Blockly workspaces. Here, we formalize the structure of *Blockly toolboxes* (and some additional aspects) in the form of an abstract grammar. Listing 1 shows the Rascal Algebraic Data Type (ADT) `Toolbox`, capturing the abstract structure of a *Blockly toolbox*.

A `Toolbox` consists of a list of `Sections`. Each `Section` has a category name, a color, and contains a list of block types (`Block`). A `Block` also has a name (e.g., “if-then”), a type name (e.g., “Statement”), and a list of messages.

The remaining arguments of the `Block` constructor are optional (because they have assigned a default value) and are used to further configure the block type. For instance, the `Ref` arguments configure the block's connectivity, where a `Ref` refers to another block-type (identified by name). The `extensions` and `mutator` argument allows hooking into native JavaScript code. The Boolean `inputsInline` toggles whether input elements should be shown inline. The other arguments should be self-explanatory.

The `Message` type captures the core syntactic mechanism of a block. It contains a format string where `%i` indicates a placeholder for every argument in the `args` list. For instance, an if-statement could have the format string “if %1 {%2}”, with two arguments, one of type `input` (to enter a conditional expression) and one of type `statement` to allow inserting a body.

## 3 Kogi

### 3.1 Introduction

Kogi [32] is a tool for describing and deriving block-based environments from context-free grammars using the Rascal [29] metaprogramming language and the Blockly library. In this section, we explain and illustrate how we derive a block-based environment from a context-free grammar.

The left-hand side of Figure 5 shows a simple DSL grammar for defining state machines, written using Rascal's built-in grammar formalism. It consists of a few rules introducing

```
data Toolbox = toolbox(list[Section] sections);

data Section
  = section(str category, Color color, list[Block] blocks);

data Block = block(str name, str \type,
  list[Message] messages, Ref output = none(),
  Ref prev = none(), Ref next = none(),
  Color color = none(), str tooltip = "",
  str helpUrl = "", list[str] extensions = [],
  str mutator = "", bool inputsInline = false);

data Message
  = message(str format, list[Arg] args);

data Arg
  = arg(str name, Type \type, Arg alt=none())
  | none();

data Type
  = value(list[str] check = [])
  | statement(list[str] check = [])
  | dummy()
  | input(str text, bool spellcheck = true)
  | dropdown(list[str] options)
  | checkbox(bool checked = false)
  | color(str color)
  | number(num \value, Range range = none())
  | angle(num angle)
  | variable(str variable, list[str] variableTypes = [])
  | date(datetime date)
  | label(str text, str class = "")
  | image(str src, int width, int height, str alt = "");

data Ref
  = block(str \type) | none();

data Range
  = range(num min, num max, num precision) | none();

data Color
  = rgb(str rgb) | hsv(int hsv) | none();
```

**Listing 1.** Algebraic data type modeling Blockly toolboxes.

a nonterminal (e.g., `Machine`), where each rule consists of several labeled productions (e.g., `State` has a single production, labeled `state`). Nonterminals can be start nonterminals (e.g., `Machine`), context-free nonterminals (e.g., `State`), or lexical nonterminals (e.g., `Id`). Rascal employs (generalized) scannerless parsing, so there is no essential distinction between context-free and lexical syntax, except in the way layout (whitespace, comments, etc.) is handled.

Kogi exploits Rascal's facilities for type reflection since each nonterminal represents a type of a parse tree; it can be applied to inspect and process grammars as values. A value



```

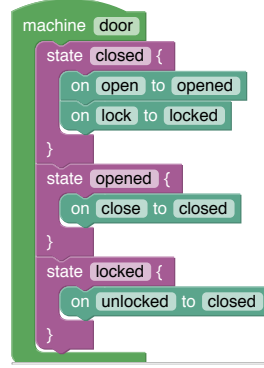
start syntax Machine
  = machine: "machine" Id name
    State* states;

syntax State
  = state: "state" Id id "{"
    Trans* transitions
  "}";

syntax Trans =
  trans: "on" Id on "to" Id to;

lexical Id = id: [a-zA-Z]+;

```



**Figure 5.** State machine grammar (left) and an example state machine (right) using the Kogi generated block-based environment.

representation of a type is acquired using the `#` operator. For instance, consider the following Rascal snippet:

```
type[Machine] typeOfMachine = #Machine;
```

The variable `typeOfMachine` will contain a structured meta-representation of the grammar defined in Figure 5, and will have type `type[Machine]`.

Note that a block-based editor essentially is a projectional editor for manipulating the abstract syntax of a language. However, for our purpose, the use of concrete syntax definitions is essential, since the keywords, operators, parentheses, etc. present in the grammar productions allow us to automatically derive the format strings (see above) required to render blocks in an informative way.

Kogi operates in three steps:

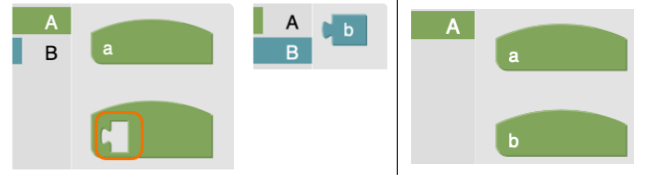
1. Analyze and preprocess a context-free grammar and transform it into a value of type `Toolbox`;
2. Run customization code, if any, provided by the language engineer, to supplant the result of step 1 with additional information not present in the grammar (e.g., colors, tooltips, etc.);
3. Generate Blockly code from the (possibly customized) `Toolbox` value.

Below we discuss each step in more detail.

### 3.2 Preprocessing the Grammar

Kogi first normalizes the grammar to a more straightforward form to facilitate the actual mapping to the `Toolbox` data type. This consists of two steps:

- Eliminate disambiguation constructs: Disambiguation using priority declarations or associativity, longest match for identifiers, and keyword reservation are irrelevant in a block-based editor, so we normalize the grammar not to have such directives.
- Inline chain rules: Chain rules introduce additional non-terminals that would introduce blocks that “do



**Figure 6.** Effect of chain rules (left) vs no chain rules (right).

nothing” except injecting one kind of element into the type of another. We inline chain rules to prevent the generation of such blocks, and remove nonterminals that have become unreachable.

The most important step here is eliminating chain rules, since it directly affects the usability of the generated environment. To illustrate the effect, consider the following grammar:

```

start syntax A = p1: "a" a | p2: B;
syntax B = b: "b";

```

A has two productions ( $p_1$  and  $p_2$ ), and B has a single production (b). Production  $p_2$  in A is a chain rule.

Therefore, we want to replace this production with B’s production (b). Additionally, replacing production rules is not enough, since it only replaces productions, but it does not remove unreachable productions from the start symbol. Hence, after removing production rules, Kogi checks unreachable nonterminal symbols; when Kogi finds an unreachable nonterminal, it is deleted from the grammar (e.g., nonterminal B becomes unreachable after removing production  $p_2$  from A).

The effect is illustrated in Figure 6. The chain rule block is highlighted with a red square on the left-hand side of the figure. The chain rule block links the nonterminal A with B. After including B’s production directly into A, the environment is much simpler, as shown in the right-hand side of Figure 6.

### 3.3 From Grammar to Toolbox

Kogi’s transformation between context-free grammars and Toolbox is relatively straightforward, and can be summarized as follows:

- Map every nonterminal  $N$  to a section named  $N$  and category name  $N$ .
- Map every production (labeled  $l$ ) of a nonterminal  $N$  to a block in the  $N$ -section:
  - name the block  $l$
  - set its type to  $N/l$  and let its output refer to  $N$
  - add a message with a format consisting of all literals interleaved with  $\%i$  placeholders for each non-literal symbol between them.
  - for each symbol  $S_i$  in the production that is not a literal, if it is a:

- \* lexical: if known, add an argument of the corresponding type, otherwise use `text`; set output to refer to type  $S_i$ , and set `inputsInline` to `true`.
- \* list of  $S$ : schedule all  $S$ -blocks to have `prev` and `next` to refer to  $S$ ; add a `statement` argument to the  $l$ -block to get vertical nesting;
- \* nonterminal: add a `value` argument for horizontal alignment, and a check for  $S_i$ .

In other words, each nonterminal corresponds to a category, and each production of a nonterminal ends up as a block type in that category. Blocks are given a unique name based on the nonterminal and production label. The format string of messages is derived from the literals in the production, and the argument list derives from the non-literal symbols, such as lexicals, context-free nonterminals, and lists. Note that list symbols (e.g., `State*`) trigger vertical stacking by setting the `prev` and `next` references of the element type (e.g., `State`).

In terms of the example of Figure 5, the mapping of productions to block types is shown in Table 1. The start symbol `Machine` is mapped to a “top” block, indicated by the arc on top, which means it cannot be nested inside any other block.

When a production contains a lexical element, Kogi applies name-based heuristics to map a terminal symbol to one of the built-in value blocks of Blockly. This heuristic is summarized in Table 2. Blockly has different visual built-in fields for different data types, such as numbers, strings, and images. However, in context-free grammars, there are no constraints for defining the terminal symbols of a language because they are described using arbitrary regular expressions. Thus, Kogi transforms every terminal symbol either into a *value block* or an inline field. When the terminal symbol is one of the built-in data types, Kogi creates an inline field, or a value block otherwise. For instance, the lexical `id` in Figure 5, is used to capture identifiers. According to Table 2, it is mapped into an inline field of type *Id value*.


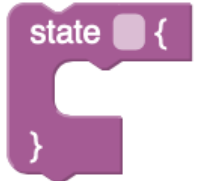

To illustrate the usage of lists within a production rule, consider the `State` production in Figure 5. This production has several literals (`state`, `{`, and `}`, a single lexical element (`Id`), and a list of transitions `Trans`). When Kogi finds a list or a separated list<sup>2</sup> it creates a *statement block* with both top and bottom connections. As shown in Table 1, both `Machine` and `State` blocks allow vertical nesting of states and transitions, respectively.

### 3.4 Customization







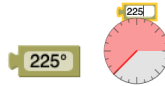
Kogi transforms a context-free grammar to a `Toolbox` value, which it then compiles to Blockly HTML and JavaScript. However, not all relevant information for a usable block-based environment is present in the grammar. Some aspects

<sup>2</sup>Separated lists are regular grammar symbols in Rascal; for instance `{Stm " ; "}` captures a list of zero or more statements (`Stm`), separated by semicolons. Since separators have no purpose in block-based environments, Kogi treats separated lists as ordinary lists.

**Table 1.** Correspondence between productions and blocks.

Type	Production	Block
Machine	"machine" Id State*	
State	"state" Id "{" Trans* "}"	
Trans	"on" Id "to" Id	

**Table 2.** Heuristics to map lexical symbols to block shapes.

Lexical name	Block
String value	
Id value	
Integer value	
Float value	
Image value	
Boolean value	
Angle value	

can be addressed through heuristics (e.g., color schemes to choose colors for categories), but in the end, it is important that language designers can customize the generated environment.

However, the resulting environment might require a few enhancements or changes depending on each use case. To address this, Kogi also supports the customization of blocks. The customization mechanism allows developers to adapt both the language’s blocks and its toolbox.

The intermediate model described by the `Toolbox` type provides the entry point for such customization. Kogi first produces a default `Toolbox`, and then optionally, the language designer can transform or change the toolbox structure according to their wishes. For instance, to assign tool-tips and colors, better labels, etc. And only *then* the Blockly JavaScript code is generated.

```

<block type="Machine/machine" id="*S8kNTF=$db4[yf36Gm;">
  <field name="id">process</field>
  <statement name="states">
    <block type="State/state" id="LMF:#e+qE{[{'wk+VOGP">
      <field name="id">idle</field>
      <statement name="transitions">
        <block type="Trans/transition" id="2=HGyRBknSM^0L3">
          <field name="on">idle</field>
          <field name="to">busy</field>
        </block>
      </statement>
    </block>
  </statement>
</block>

```

**Listing 2.** Blockly XML representation of a state.

### 3.5 Execution

Besides, to create a block-based UI for a language, Kogi also allows users to reuse language components, such as parsers and interpreters. To reuse these language components, Kogi maps an XML representation of the programs, obtained directly from the block-based editor, to a native AST structure. Because Kogi uses the names of nonterminals and productions label to define the toolbox, these names can again be used to reflectively map this XML structure back to an AST datatype that uses the same names.

For instance, a suitable abstract syntax definition for the state machine example of Figure 5 would be:

```

data Machine = machine(str name, list[State] states);
data State = state(str id, list[Trans] transitions);
data Trans = trans(str on, str to);

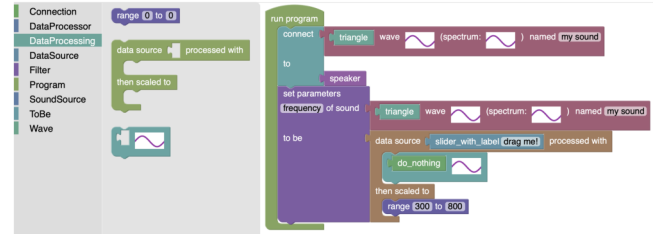
```

Each type corresponds to a nonterminal, and each constructor to a named production. Lexicals are mapped to constructors with string arguments (*str*).

Listing 2 shows an (excerpt) of an XML AST returned by the generated Blockly environment for the state machine language of Figure 5. Using the names in the type and name attributes, this can be transformed into a value of type (in this case) *state*. Thus, the type value can be used by language processors of the language.

## 4 Case Studies

We used Kogi to generate block-based environments for four different languages, namely, the state machine language discussed above, Sonification Blocks, Pico, and QL. These languages were implemented using Rascal and are available on GitHub<sup>3</sup>. Below we briefly discuss the latter three languages.



**Figure 7.** Kogi block-based version of Sonification Blocks.

### 4.1 Sonification Blocks

Sonification Blocks [4] is a programming language for teaching students basic concepts of sound production, programming, and connection of data flows. This language is offered as a custom-made block-based environment.

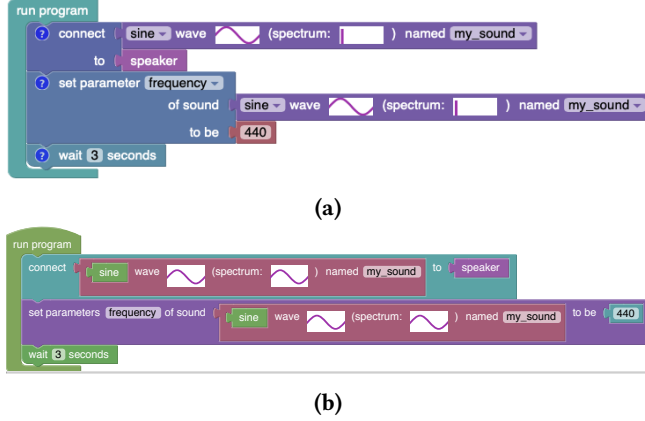
We have manually reverse engineered Sonification Blocks and implemented a Rascal grammar that captures the language's syntax. Then this grammar was input to Kogi to create the block-based environment shown in Figure 7. Figure 8 shows an original Sonification Blocks program compared to the same program in the Kogi-generated environment.

It can be observed that both programs (Figure 8) are quite similar. However, there are some differences. For instance, the children of the *run program* block do not have the same layout. The generated version rendered them in a single line. Moreover, the *connect* block in the generated environment does not allow the user to select a value from a dropdown list; instead, it offers all the options as standalone blocks (e.g., *sine*). The same difference is found in the last field (named of the same block), in which users must write the variable's name manually. Finally, the images for waves and spectrum are hard-coded in Kogi's version, meaning that they do not change as the user changes other values. While the first differences could be considered cosmetic, the second category is more significant, since the display of dynamic sine waves is essential for the programmer experience.

As explained in Section 3.4, Kogi allows developers to customize the generated block-based environment. Therefore, we will customize the generated environment by changing a block's color and defining the toolbox categories. To customize a block-based environment, developers must create references to the blocks they want to customize and then define the categories in which blocks will be grouped. Each block is referenced by the production's label. Listing 3 shows how to customize the Sonification Blocks' toolbox.

First, we create references to four blocks: *initial*, *sound*, *speaker*, and *slider*. Based on these references, we change the color of the *initial* block; the other blocks remain unchanged. Moreover, we created three custom categories: *Start*, *Connection*, and *Sources*. If a block is not assigned to any of the custom-defined categories, Kogi sets them into an *Unassigned* default category. The resulting custom Sonification Blocks environment is shown in Figure 9.

<sup>3</sup><https://github.com/cwi-swat/kogi-examples>



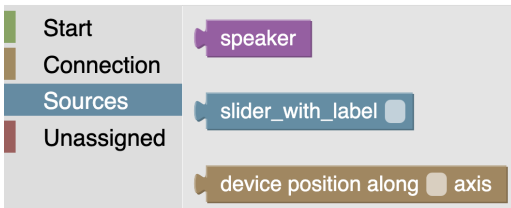
**Figure 8.** Original Sonification Blocks program [3] (a) and Kogi version (b).

```
Toolbox customizeToolbox() {
  // Blocks references
  initial = block("initial", colour = hsv(360));
  sound = block("sound");
  speaker = block("speaker");
  slider = block("slider");

  // Toolbox sections
  initSec = section("Start", hsv(90), [initial]);
  connection = section("Connection", hsv(0), [sound]);
  dataSource = section("Sources", hsv(200),
    [speaker, slider]);

  return toolbox([initSec, connection, dataSource]);
}
```

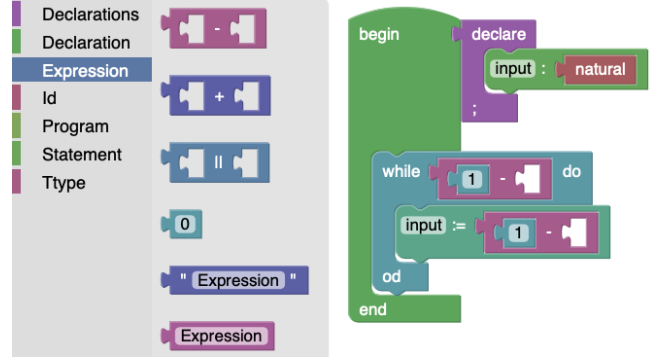
**Listing 3.** Customization of Sonification Blocks.



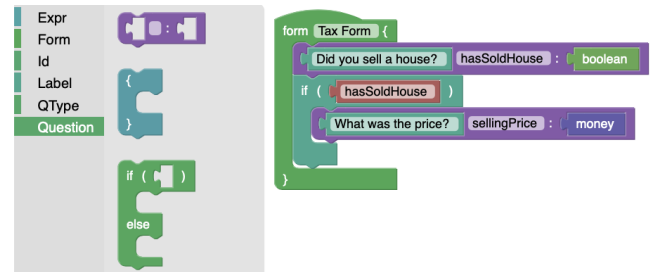
**Figure 9.** Customized version of Sonification Blocks.

## 4.2 Pico

Pico is a toy programming language, like the WHILE language, often used in programming language semantics textbooks. An implementation of Pico is available as part of the Rascal standard library [39]. To create the block-based interface for Pico, we used the existing Rascal grammar for Pico, and used it as input to Kogi; the resulting environment is shown in Figure 10. Kogi allows us to reuse existing language components, not only the grammar for deriving the block-based UI, but the interpreter for executing programs.



**Figure 10.** Block-based environment for Pico.



**Figure 11.** Block-based environment for QL.

## 4.3 QL

QL is a DSL for defining interactive questionnaires, and it has been used to benchmark and evaluate language workbenches [15]. QL is interesting to be used within a block-based environment because it is not a programming language, which means that the target users might be domain experts who have limited or no programming experience. Therefore, block-based environments could be more natural to use for this kind of end-user [50, 52] due to the use of natural language labels on blocks, colors, shapes, and the interaction with the environment (drag and drop).

Rascal already had an implementation for QL; thus, we used the existing implementation to obtain a block-based syntax. We took QL's concrete syntax in Rascal and used it as input for Kogi. Figure 11 shows an example of a tax questionnaire defined using the generated block-based environment. In this example, a domain-expert could have defined a simplified tax form with two questions, a single question (hasSoldHouse), and a conditional question (sellingPrice).

## 4.4 Effort

To better understand the effort of developing block-based environments, we measured the number of Source Lines of Code (SLOC) for the generated environments. This includes the number of SLOC of the grammar in Rascal and the number of SLOC of generated Blockly JavaScript and HTML. All the SLOC measurements for the generated (and the manual



implementation Sonification Blocks) of the environments were done using SonarQube [43] and Cloc [14]. The SLOC for the Rascal grammars were measured only with a configured of Cloc so that it supports Rascal.

Table 3 shows the number of SLOC for each language's grammar, the number of generated SLOCs per block-based environment, and the Sonification Blocks' handwritten SLOC. The first column shows the names of the considered languages. The next column has the number of grammar SLOCs for each language; all the grammars were written using Rascal's syntax definition formalism. The following two columns, *Generated Blockly code* and *Manual implementation*, contain the SLOC generated by Kogi for each environment and the handwritten version of Sonification Blocks.

We only included the manual implementation of Sonification Blocks because it is an existing block-based environment [3], while the others were pre-existing languages, implemented in Rascal, but without a manual implementation of a block-based environment. The Generated Blockly code column is divided into two columns, *with chain rules* and *without chain rules*. The first one represents the environment as-is, without removing chain rules and the latter is the environment where chain rules and unreachable productions are removed.

For each environment in Table 3, we calculated the number of SLOC for *HTML*, *JS*, and *Total*. The *HTML* SLOC contains a default web application in such language. Kogi's generated environment is a basic HTML app that loads required JS libraries (e.g., Blockly) and creates a basic layout to display the block-based environment and an XML representation of the current block program. The HTML app contains the definition of the toolbox as an embedded XML element.

The *JS* code represents the language blocks' specifications using Blockly's embedded JSON DSL. The *Total* column is the sum of the HTML and the JS columns. In general, the number of HTML SLOC is smaller for the generated environment than the JS SLOC because, as mentioned above, the HTML app is rather basic, while the JS contains the definition of all the blocks, and each block's definition requires around 20–30 SLOCs.

When comparing the SLOC of the environments with and without chain rules, some differences become apparent. First, for the state machine language, there is no difference, because there are no chain rules in the grammar. Comparing the results of QL and Pico, however, there is a decrease in the number of SLOC in the environments where chain rules have been removed.

The results of Sonification Blocks, however, show a different picture. In this case, the number of SLOC increases in the environment without chain rules. This behavior is caused by inlining chain rules: If a chain rule is used in multiple places in the grammar, it will be inlined multiple times. As a result, duplicate blocks are created in different categories.

#### 4.5 Effect of Chain Rule Elimination

To evaluate the effect of chain rule elimination, we manually calculated the number of toolbox categories and blocks per language for both environments, the one that contains chain rules (*Standard grammar*) and the other that does not contain chain rules (*Simplified grammar*). Table 4 shows the results for all the languages.

In the first row (State machines), as we saw in Table 3, there is no difference between the two environments. As we discussed in Section 3, removing chain rules might directly impact the number of nonterminals. This impact varies depending on how the grammar was written; and the relationships between the nonterminals involved in a chain rule. As we see from the data in Table 4, the number of categories decreases in most languages (except in state machines where no chain rules were found). Looking at the number of blocks, in two (QL and Pico) out of the four cases, there was also a reduction in the number of blocks. Nonetheless, in two cases (State machines and Sonification Blocks), there was no reduction nor increase in the number of blocks. As discussed earlier, the state machine language does not have chain rules, yet Sonification Blocks does have chain rules, as can be seen in reducing the toolbox' categories, but in the latter case, additional, duplicated blocks were generated, which causes the numbers to be the same.

#### 4.6 Discussion

As we explained through the paper, Kogi uses Rascal as a platform for developing and generating the resulting block-based environments. This fact shows that using a Language Workbench (LWB) syntax definition formalism is possible. Moreover, as shown by Kogi, we can use these formalisms for describing and creating block-based environments. For instance, Kogi uses an existing LWB (Rascal) for the specification of a block-based environment with Blockly as front-end. The way Kogi does it is by using context-free grammars to describe the language's syntax, and deriving a block-based environment from it. However, as observed in the generated environments, they often require some adjustments. Particularly, when comparing the generated version of Sonification Blocks and its manual implementation, we notice that, as expected, the latter contain some tweaks to improve the user's experience. This is a common trade-off between an ad-hoc and a generated solution like the one offered by Kogi. Since generated solutions might not fit all use cases, Kogi offers some degree of block customization, as explained in Section 3. For instance, in Sonification Blocks (Section 4), we used Rascal for describing the language's syntax; based on this definition, we derived a block-based language (Figure 8). Likewise, the specification of the language was done using a context-free grammar. As we showed with the four case studies, the information contained in context-free grammars is expressive enough to create a block-based environment.

**Table 3.** Lines of code (SLOC) numbers of Kogi generated environments (written and generated).

Languages	Grammar (SLOC)	Generated Blockly code (SLOC)						Manual Implementation (SLOC)		
		With chain rules			Without chain rules					
		HTML	JS	Total	HTML	JS	Total	HTML	JS	Total
State machine	15	35	112	147	35	112	147		-	
Sonification	75	85	769	854	79	874	953	1752	536	2288
QL	54	69	811	880	64	733	797		-	
Pico	39	55	408	463	52	389	441		-	

**Table 4.** Number of categories and blocks per language.

Languages	Standard grammar		Simplified grammar	
	# Cats.	# Blocks	# Cats.	# Blocks
State machines	3	3	3	3
Sonification	12	35	9	35
QL	6	31	5	28
Pico	7	15	6	14

Moreover, Kogi supports user-defined customization. Kogi's customization mechanism allows developers to make modifications at both the toolbox and the block level. The first allows users to create their own set of toolbox categories and group blocks within these categories, while the second lets developers define or tweak single blocks for their needs. Thus, Kogi's customization mechanism allows developers to adapt a generated solution to fit their needs.

As we observed in Table 4, eliminating chain rules in a grammar reduces the number of toolbox's categories and blocks. However, a reduction in these numbers does not guarantee an improvement in the end-users' editing experience. We tried out both environments (with and without chain rules), and often the environments with chain rules require end-users to add extra blocks to their programs; further research is needed to measure the impact of removing chain rules in terms of the environment's usability. Therefore, we do not have enough quantitative or qualitative results to conclude that removing chain rules impacts these environments' usability. Nonetheless, we noticed some differences in the editing experience when we removed the 'chain blocks' from the block-based environment.

## 5 Further Directions

**Block Grammars.** Kogi applies heuristics to obtain a usable default Block layout based on the structure of a context-free grammar. After mapping the grammar to the `Toolbox` data type, language designers can customize some of the aspects to obtain a better user experience. Nevertheless, both the heuristics and customization hooks are relatively limited

to the potential offered by block-based UIs. An interesting direction to offer more flexibility to language designers is then to explore a "native" grammar formalism for blocks, where properties like orientation (vertical vs. horizontal), inline rendering, colors, tool-tips, etc. are first-class citizens in the grammar. Integration with a UI framework could even allow the language designer to define custom "lexical" elements, to supplant the basic set offered by frameworks like Blockly.

**Hybrid Languages.** Although Block-based languages have the potential to lower the barrier to entry to programming for end-users, at a certain level of detail, the block metaphor may break down. For instance, expressions are a widely used and well-known concept, and they are found in many languages. Pasting together expressions (especially deeply nested ones) in a block-based environment, however, can be tedious and cumbersome.

A direction to explore would thus be to support hybrid languages, where some constructs are block-based, but others, such as the aforementioned expressions, are based on parsing text-fields. In a sense, this is also how spreadsheets work: The grid is a structured editor, but the formulas are entered textually.

A further benefit of such hybrid editor could be that it emphasizes the difference between programming and configuration: Blocks for defining the high-level architecture of a system by composing components (such as machines, classes, entities, UIs, robots, etc.), – but using code editors for low-level algorithmic details.

**Error Marking.** Block-based environments provide a way to specify a program without the possibility of making syntax errors. However, most languages have consistency and well-formedness checks that go beyond pure syntax, such as type checking. Kogi-based editors support a level of reuse of existing language components. However, for type checking (or any kind of static analysis), this is currently limited to printing out errors on the console. It would be interesting to explore origin tracking [24, 48] techniques to allow highlighting such errors within the editor itself. For instance, by propagating the node identities of the XML AST produced by Blockly (as seen in Listing 2) to the native Rascal ASTs,

and using those identities to render the errors in-place using JavaScript.

**Run-Time Support.** Block-based syntax support (possibly with error marking) is concerned with static aspects of code. However, an important aspect, especially for end-users, is being able to inspect and visualize executing code. Many block-based environments (e.g., Scratch) are also live programming languages, where dynamic inputs are entered inside the IDE, and the dynamic execution can be started, interrupted, restarted, inspect, etc. Such run-time feature would require a deeper integration between the block-based front-end and the internal structures (stack frames, heaps, program counters, etc.) of the back-end. Further research is needed to investigate how far such support is possible, while still being able to reuse as much as possible of existing language artifacts.

## 6 Related Work

Block-based environments can be considered a subclass of the class of graphical or visual languages [12]. One of the main motivations of graphical languages is to make programming for beginners easier than text-based languages [27]. Kogi contributes to the research field of generating programming environments [10, 15, 21, 28, 40, 44, 47]. In the literature, we found mainly two ways of developing block-based environments: through libraries or extending existing block-based environments.

Begel [7] created a graphical version of Logo, a computer language developed in the 1960's by Seymour Papert et al., with the aim of lowering the barrier to entry for learners. Valarte [42] designed and developed a framework for creating graphical block programming systems through a specification in XML format. Blockly [36] offers an API for creating block-based UIs; they offer two APIs for the block's definition, one in JavaScript, and the other using JSON.

Extending existing block-based environments is also a common practice to develop block-based environments. Tamiias et al. [46] extended Blockly@rduino to create B@SE, a block-based environment to ease the transition from blocks to text-based programming. Similarly, Nergaard [35] created a block-based policy editor for XACML by extending Scratch. Kyfonidis et al. [31] extended OpenBlocks to create a block-based version of the C programming language.

Kurihara et al. [30] proposed a programming environment for visual DSLs that uses code generators. The code generators are used to generate text-based code from the block-based representation. Kogi offers a similar approach, yet Kogi is integrated within an LWB, which lets developers define all the language's aspects. Moreover, Kogi supports can be used to build block-based UIs on top of existing languages developed in a LWB; as a result, existing text-based languages can benefit from having an additional block-based UI.

## 7 Conclusions and Future Work

Block-based environments offer a different UI for interacting with code, in which writing a program becomes a matter of dragging and dropping jigsaw-like blocks. This type of environment has become popular due to the benefits they offer to end-users: no risk of syntax errors, easy discoverability, labels in natural language, etc. Moreover, this type of environment is being used in different domains, ranging from education to robot programming.

Nevertheless, the implementation of block-based languages requires a lot of effort, because high-level language workbench support is currently lacking. Libraries like Blockly help developers to create the front-end of block-based languages, but still require low-level, framework-specific programming.

In this paper we have presented Kogi, as a step towards first-class support for block-based language as part of language workbenches (in this case Rascal) by deriving block-based environments from context-free grammars. We have analyzed the anatomy of block-based environments by dissecting Google's Blockly framework (Section 2), and formalized it as an abstract syntax for Blockly toolboxes. Kogi takes a context-free grammar and transforms it to a Blockly AST which is then compiled to the required Blockly JavaScript code. The grammar is analyzed to obtain reasonable defaults for the layout and categorization of the resulting blocks. To improve the usability of the generated environment, Kogi applies a number of simplifications to the grammar, to avoid generation of spurious blocks types. Blockly-based environments export the program as an XML AST, which can be mapped back to a native Rascal AST structure, which is suitable for further processing (interpretation, code generation, etc.).

We have used Kogi to create block-based environments for four languages, namely a DSL for State machines, an existing language for sound configuration, Sonification Blocks, a DSL for questionnaires QL, and a simple programming language, Pico (Section 4). The generated environments are evaluated in terms of effort (Section 4.4) and toolbox complexity (Section 4.5).

Kogi represents the first step to integrate block-based syntax with language workbenches. The resulting environments are usable, and may be supported by (pre-)existing language components. Nevertheless, further research is required to provide a more native formalism to define, configure, and customize block-based environments to offer maximal flexibility to language designers, investigating hybrid environments combining both block-based elements and textual syntax, and how to support more dynamic aspects of a language, such as debugging, providing dynamic inputs, and live programming.



## Acknowledgments

We want to thank Jack Atherton for sharing with us the source code of Sonification Blocks.

## References

- [1] Adaptavist. 2019. *AutoBlocks for Jira*. <https://docs.adaptavist.com/autoblocks/jira/server/latest/>
- [2] Saksham Aggarwal, David Anthony Baau, and David Bau. 2015. A blocks-based editor for HTML code. *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015* (2015), 83–85. <https://doi.org/10.1109/BLOCKS.2015.7369008>
- [3] Jack Atherton and Paulo Blikstein. 2017. *Define blocks*. Retrieved July 20, 2020 from <https://ccrma.stanford.edu/~lja/sonification/>
- [4] Jack Atherton and Paulo Blikstein. 2017. Sonification Blocks: A Block-Based Programming Environment For Embodied Data Sonification. In *Proceedings of the 2017 Conference on Interaction Design and Children* (Stanford, California, USA) (*IDC '17*). Association for Computing Machinery, New York, NY, USA, 733–736. <https://doi.org/10.1145/3078072.3091992>
- [5] David Bau. 2015. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges* 30, 6 (2015), 138–144.
- [6] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. <https://doi.org/10.1145/3015455>
- [7] Andrew Begel. 1996. *LogoBlocks: A Graphical Programming Language for Interacting with the World*. Master's thesis. Massachusetts Institute of Technology, Media Laboratory.
- [8] Benedikt Breuch and Martin Fislake. 2020. First Steps in Teaching Robotics with Drones. In *Robotics in Education*. Springer International Publishing, Cham, 138–144.
- [9] Martin Cápáy and Nika Klimová. 2019. Engage your students via physical computing! *IEEE Global Engineering Education Conference, EDUCON April-2019* (2019), 1216–1223. <https://doi.org/10.1109/EDUCON.2019.8725101>
- [10] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Evelyn Duesterwald, and Jurgen Vinju. 2009. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. *SIGPLAN Not.* 44, 10, 191–206. <https://doi.org/10.1145/1639949.1640104>
- [11] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello B. 2019. My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor. 11553 (2019), 18–33. <https://doi.org/10.1007/978-3-030-24781-2>
- [12] Enrique Coronado, Fulvio Mastrogiovanni, Bipin Indurkha, and Gentiane Venture. 2020. Visual Programming Environments for End-User Development of intelligent and social robots, a systematic review. *Journal of Computer Languages* 58 (2020), 100970. <https://doi.org/10.1016/j.cola.2020.100970>
- [13] Chris S. Crawford and Juan E. Gilbert. 2019. Brains and blocks: Introducing novice programmers to brain-computer interface application development. *ACM Transactions on Computing Education* 19, 4 (2019). <https://doi.org/10.1145/3335815>
- [14] Al Danial. 2006. *Cloc*. Retrieved July 20, 2020 from <https://github.com/AIDanial/cloc>
- [15] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24 – 47. <https://doi.org/10.1016/j.cl.2015.08.007>
- [16] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) (*OOPSLA '10*). Association for Computing Machinery, New York, NY, USA, 307–309. <https://doi.org/10.1145/1869542.1869625>
- [17] Gilad Gome, Julian Waksberg, Andrey Grishko, Iddo Yehoshua Wald, and Oren Zuckerman. 2019. OpenLH: Open Liquid-Handling System for Creative Experimentation with Biology. In *Proceedings of the Thirteenth International Conference on Tangible, Embedded, and Embodied Interaction* (Tempe, Arizona, USA) (*TEI '19*). Association for Computing Machinery, New York, NY, USA, 55–64. <https://doi.org/10.1145/3294109.3295619>
- [18] Google. 2020. *Blockly*. Retrieved July 10, 2020 from <https://developers.google.com/blockly>
- [19] Google. 2020. *Define blocks*. Retrieved July 13, 2020 from [https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks#block\\_inputs](https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks#block_inputs)
- [20] Google. 2020. *Define blocks*. Retrieved July 13, 2020 from [https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks#block\\_output](https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks#block_output)
- [21] Jan Heering and Paul Klint. 2000. Semantics of Programming Languages: A Tool-oriented Approach. *SIGPLAN Not.* 35, 3, 39–48. <https://doi.org/10.1145/351159.351173>
- [22] Charlotte Hill, Hilary A. Dwyer, Tim Martinez, Danielle Harlow, and Diana Franklin. 2015. Floors and flexibility: Designing a programming environment for 4th–6th grade classrooms. *SIGCSE 2015 - Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), 546–551. <https://doi.org/10.1145/2676723.2677275>
- [23] Robert Holwerda and Felienne Hermans. 2018. A usability analysis of blocks-based programming editors using cognitive dimensions. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC October* (2018), 217–225. <https://doi.org/10.1109/VLHCC.2018.8506483>
- [24] Pablo Inostroza, Tijs van der Storm, and Sebastian Erdweg. 2014. Tracing Program Transformations with String Origins. In *Theory and Practice of Model Transformations*, Davide Di Ruscio and Dániel Varró (Eds.). Springer International Publishing, 154–169.
- [25] Afnan Islam, Thajid Ibna Rouf Uday, Nazib Ahmad, Md Toriqul Islam, Amit Ghosh, Sadia Kamal, Tanzir Ahommed, Mazharul Islam Leon, and Ehsan Ahmed Dhruvo. 2019. EduBot: An Educational Robot for Underprivileged Children. *2019 International Conference on Automation, Computational and Technology Management, ICACMTM 2019* (2019), 232–236. <https://doi.org/10.1109/ICACMTM.2019.8776756>
- [26] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) (*OOPSLA '10*). Association for Computing Machinery, New York, NY, USA, 444–463. <https://doi.org/10.1145/1869459.1869497>
- [27] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (June 2005), 83–137. <https://doi.org/10.1145/1089733.1089734>
- [28] P. Klint. 1993. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (April 1993), 176–201. <https://doi.org/10.1145/151257.151260>
- [29] P. Klint, T. v. d. Storm, and J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 168–177.
- [30] Azusa Kurihara, Akira Sasaki, Ken Wakita, and Hiroshi Hosobe. 2015. A Programming Environment for Visual Block-Based Domain-Specific Languages. *Procedia Computer Science* 62 (2015), 287 – 296. <https://doi.org/10.1016/j.procs.2015.08.452> Proceedings of the 2015 International



- Conference on Soft Computing and Software Engineering (SCSE'15).
- [31] Charalampos Kyfonidis, Nektarios Mousmoutzis, and Stavros Christodoulakis. 2017. Block-C: A block-based programming teaching tool to facilitate introductory C programming courses. *IEEE Global Engineering Education Conference, EDUCON* April (2017), 570–579. <https://doi.org/10.1109/EDUCON.2017.7942903>
- [32] Mauricio Verano Merino and Tijs van der Storm. 2020. cwi-swat/kogi: Kogi 0.1.0. (Sep 2020). <https://doi.org/10.5281/zenodo.4033220>
- [33] Jens Mönig and Brian Harvey. 2018. *Snap!* Retrieved August 22, 2018 from <https://snap.berkeley.edu>
- [34] Luke Moors and Robert Sheehan. 2017. Aiding the Transition from Novice to Traditional Programming Environments. In *Proceedings of the 2017 Conference on Interaction Design and Children* (Stanford, California, USA) (IDC '17). Association for Computing Machinery, New York, NY, USA, 509–514. <https://doi.org/10.1145/3078072.3084317>
- [35] H. Nergaard, N. Ulltveit-Moe, and T. Gjøsæter. 2015. A scratch-based graphical policy editor for XACML. In *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. 1–9.
- [36] E. Pasternak, R. Fenichel, and A. N. Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B/B)*. 21–24.
- [37] Evan W. Patton, Michael Tissenbaum, and Farzeen Harunani. 2019. *MIT App Inventor: Objectives, Design, and Development*. Springer Singapore, Singapore, 31–49. [https://doi.org/10.1007/978-981-13-6528-7\\_3](https://doi.org/10.1007/978-981-13-6528-7_3)
- [38] Thomas W. Price and Tiffany Barnes. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) (ICER '15). Association for Computing Machinery, New York, NY, USA, 91–99. <https://doi.org/10.1145/2787622.2787712>
- [39] Rascal. 2017. *Pico*. Retrieved July 20, 2020 from <http://tutor.rascal-mpl.org/Recipes/Recipes.html#/Recipes/Languages/Pico/Pico.html>
- [40] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. Association for Computing Machinery, New York, NY, USA, 42–48. <https://doi.org/10.1145/800020.808247>
- [41] Mitchel et al. Resnick. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67.
- [42] Ricarose Vallarta Roque. 2007. *OpenBlocks: An Extendable Framework for Graphical Block Programming Systems*. Master's thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.
- [43] SonarSource SA. 2008. *SonarQube*. Retrieved July 22, 2020 from <https://www.sonarqube.org>
- [44] Emma Söderberg and Görel Hedin. 2011. Building Semantic Editors Using JastAdd: Tool Demonstration. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications* (Saarbrücken, Germany) (LDTA '11). Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. <https://doi.org/10.1145/1988783.1988794>
- [45] Andrew Stratton, Chris Bates, and Andy Dearden. 2017. Quando: Enabling Museum and Art Gallery Practitioners to Develop Interactive Digital Exhibits. In *End-User Development*. Springer International Publishing, Cham, 100–107.
- [46] Alexander G. Tamiliadis, Theodoros J. Themelis, Theodoros Karvounidis, Zacharenia Garofalaki, and Dimitrios Kallergis. 2017. B@SE: Blocks for @rduino in the Students' educational process. *IEEE Global Engineering Education Conference, EDUCON* April (2017), 910–915. <https://doi.org/10.1109/EDUCON.2017.7942956>
- [47] Mark G.J. van den Brand, Arie van Deursen, Jan Heering, Hayco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3 – 8. [https://doi.org/10.1016/S1571-0661\(04\)80917-4](https://doi.org/10.1016/S1571-0661(04)80917-4) LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).
- [48] A. van Deursen, P. Klint, and F. Tip. 1993. Origin tracking. *Journal of Symbolic Computation* 15, 5 (1993), 523 – 545. [https://doi.org/10.1016/S0747-7171\(06\)80004-0](https://doi.org/10.1016/S0747-7171(06)80004-0)
- [49] David Weintrop. 2019. Block-based programming in computer science education. , 22–25 pages. <https://doi.org/10.1145/3341221>
- [50] David Weintrop, Afsoon Afzal, Jean Salac, Patrick Francis, Boyang Li, David C. Shepherd, and Diana Franklin. 2018. 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* (2018), 1–12. <https://doi.org/10.1145/3173574.3173940>
- [51] David Weintrop, David C. Shepherd, Patrick Francis, and Diana Franklin. 2017. Blockly goes to work: Block-based programming for industrial robots. , 29–36 pages. <https://doi.org/10.1109/BLOCKS.2017.8120406>
- [52] David Weintrop and Uri Wilensky. 2015. 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* (Boston, Massachusetts) (IDC '15). Association for Computing Machinery, New York, NY, USA, 199–208. <https://doi.org/10.1145/2771839.2771860>
- [53] David Weintrop and Uri Wilensky. 2017. Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments. In *Proceedings of the 2017 Conference on Interaction Design and Children* (Stanford, California, USA) (IDC '17). Association for Computing Machinery, New York, NY, USA, 183–192. <https://doi.org/10.1145/3078072.3079715>
- [54] David Weintrop and Uri Wilensky. 2018. How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction* 17 (2018), 83–92. <https://doi.org/10.1016/j.ijcci.2018.04.005>
- [55] Sharon Zhou, Ivy J Livingston, Mark Schiefsky, Stuart M Shieber, and Krzysztof Z Gajos. 2016. Ingenium: Engaging Novice Students with Latin Grammar. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (2016), 944–956. <https://doi.org/10.1145/2858036.2858239>