Getting Grammars into Shape for Block-Based Editors

Mauricio Verano Merino
Eindhoven University of Technology
CWI
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.de

Tijs van der Storm
CWI
University of Groningen
Amsterdam, The Netherlands
storm@cwi.nl

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.de

Jurgen J. Vinju
Eindhoven University of Technology
CWI
Amsterdam, The Netherlands
Jurgen.Vinju@cwi.nl

Abstract

Block-based environments are visual programming environments that allow users to program by interactively arranging visual jigsaw-like blocks. They have shown to be helpful in several domains but often require experienced developers for their creation. Previous research investigated the use of language workbenches to generate block-based editors based on grammars, but the generated block-based editors sometimes provided too many unnecessary blocks, leading to verbose environments and programs. To reduce the number of interactions, we propose a set of transformations to simplify the original grammar, yielding a reduction of the number of (useful) kinds of blocks available in the resulting editors. We show that our generated block-based editors are improved for a set of observed aesthetic criteria up to a certain complexity. As such, analyzing and simplifying grammars before generating block-based editors allows us to derive more compact and potentially more usable block-based editors, making reuse of existing grammars through automatic generation feasible.

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

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

ACM Reference Format:
Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3486608.3486908

SLE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9111-5/21/10.
https://doi.org/10.1145/3486608.3486908

1 Introduction

Block-based environments have become popular thanks to their ease of use, especially for end-users [3]. A block-based environment is a visual interactive programming environment, in which language constructs are represented by jigsaw-like puzzle pieces, called blocks. Blocks have different visual cues, for instance, their shape, color, or connections. These cues help users to understand how different blocks (language constructs) can be snapped together to create valid programs. Benefits of block-based interfaces include the what-you-see-is-what-you-get (WYSIWYG) programming experience and avoidance of syntax errors [25, 28, 34, 35].

An example of a block-based environment is shown in Figure 1. It consists of a *palette* (left part of Figure 1) that contains all the language constructs that can be used to create programs; the *canvas* (middle-part of Figure 1), where users create their programs by dragging and dropping blocks from the palette into the canvas; and an optional *stage* (right part of Figure 1) that is used to display output of a program's execution. Block-based environments have been used in different domains across different disciplines (e.g., Computer Science, Software Engineering, Education, Science, Music, and Art) [24].

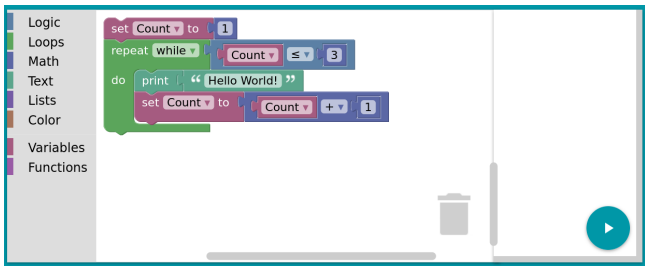


Figure 1. Block-based editor created using Google Blockly.

Block-based editors can be constructed in a variety of ways, ranging from programming from scratch, using Domain-Specific Languages (DSLs) for block definition, or visual languages. Most of these require considerable overhead or boilerplate, involving various technologies and frameworks [24].

Earlier research proposed a tool named Kogi for deriving block-based editors from declarative Context-Free Grammars (CFGs) [33]. While Kogi enabled automating most of the effort in constructing block-based editors, the usability of the derived block-based editors is limited as the derivation mechanism followed the exact structure of the input grammar. In this paper, we extend and improve Kogi's approach by analyzing the input grammar and applying structural changes to the grammar to produce block-based editors that follow a set of aesthetic guidelines we establish in Section 3.

The contributions of this paper are as follows:

- an analysis and a set of simplification rules of CFGs to improve the usability of generated block-based editors (Section 3).
- an extension of Kogi, S/Kogi, that implements the described simplification rules (Section 4).
- an evaluation that demonstrates the impact of the simplification rules for deriving block-based editors using six different languages, including Java and JavaScript (Section 5).

We continue this paper with a discussion of limitations and trade-offs, particularly considering the type and complexity of the grammars, as well directions for future work (Section 6). Finally, we present related work (Section 7), and conclude our paper (Section 8).

2 Generating Block-Based Editors from Grammars

In this section, we summarize Kogi's method for mapping grammar rules to blocks. Then, we analyze some limitations related to Kogi's generated block-based editors, and finally, present a set of aesthetic criteria.

2.1 Mapping Top-Level Alternatives to Blocks

As mentioned in Section 1, there are three ways to develop block-based editors: implementation from scratch, extending existing block-based editors, or using libraries [24]. An alternative is presented in the work by Verano and van der Storm [33], called Kogi, which derives block-based editors from context-free grammars [23]. The resulting block-based editors use Google's Blockly library [13]. Given a grammar, Kogi analyzes it to create a mapping between grammar constructs and a generic Algebraic Data Type (ADT) that describes the elements of a block-based environment. From this ADT, Kogi derives a set of named categories containing a set of blocks, where the blocks represent all possible representations of the rules of the grammar.

Table 1. Mapping between grammar rules and blocks.

Type	Rule	Block
VarDecl	`Type Id ";"`	

Stmt	`"{" Stmt* "}"`
	`| "while" "(" Expr ")" Stmt`
	`| ...`
		

Kogi's mapping from grammars to blocks is based on a set of heuristics [33]. In the following, we will summarize Kogi's method using the example of *MiniJava* (a subset of the Java language that captures the essential object-oriented features of the full Java language [1, 7]). The implementation of this language is available on GitHub[32].

Table 1 shows two rules from the MiniJava grammar and their mapping to blocks as generated by Kogi. The first rule, `VarDecl`, consists of three symbols and acts as a block for declaring variables. Its first symbol is `Type` which is a non-terminal symbol and thus gets turned into a *value input* for the resulting block. The second symbol, `Id`, is a rule that will map to a lexical [1], which is turned into a *text field* instead of an input. The third symbol is the semicolon, which is a terminal and is added as a *label* on the block. The second rule, `Stmt`, is an excerpt from the MiniJava rule for statements. Rules that have top-level alternatives are turned into a category in the palette, where each alternative is turned into a block of that category. We show two representative examples. First, the curly braces that enclose a statement block are turned into a block where the terminals act as labels again. The `Stmt*` non-terminal, rather than producing a *value input*, is turned into a so-called *statement input*, as the star indicates that multiple statements can be added here. Second, the while loop's keyword and parenthesis are turned into *labels* again. The `Expr` identifier gets turned into a *value input*. Finally, the `Stmt` non-terminal has been used with a star in the previous rule (as shown in Table 1). Because of this, even though it is not repeated in the while-loop, we still generate a *statement input*, rather than a *value input* as the same type of block cannot have two different types of input shapes.

Kogi's main limitation is that it preserves the mapping of the original grammar exactly, mapping each rule and top-level alternative to one category or block, respectively. Since grammars are used for parsing text, grammar designers often need to add syntactic elements (such as parentheses, statement terminators, etc.) to ensure that the language is unambiguous or can be parsed efficiently. In a block-based editor, however, such textual markers are often not relevant,

[1] In Rascal, lexical symbols are like syntax non-terminals, but are not modified with interleaved layout non-terminals.

because in a block-based editor programmers manipulate Abstract Syntax Trees (ASTs) directly.

Another drawback is that Kogi takes as input general CFGs with explicit constructs for operator precedence and disambiguation as supported by the Rascal language workbench [18]. However, many grammars out there have been adapted into forms that are acceptable by parsing algorithms that require a more verbose formulation of rules, such as LL(k) or LR(k). This means the grammars contain "tricks", for instance, to avoid left-recursion, or to encode precedence using layered non-terminals. Feeding such grammars to Kogi would lead to very unbalanced and difficult-to-use block-based editors.

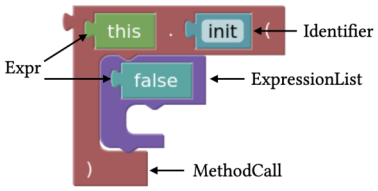
2.2 Limitations of Kogi

In the following, we will describe examples of issues we have identified in block-based editors generated by Kogi using the MiniJava grammar (Appendix A). From this, we will then derive our aesthetic criteria for blocks, following a heuristic evaluation as proposed by Nielsen and Molich [27], which allows us to identify the current limitations of Kogi and study how these limitations can be addressed. We are aware that this type of evaluation does not suggest guidelines on how to address such limitations, the purpose of this paper, however, is to show how these limitations can be mitigated by transforming the rules within a CFG to produce block-based editors that closely resemble popular, hand-crafted editors. Similar to Holwerda and Hermans in their evaluation of block-based user interfaces [15], we will apply the Cognitive Dimensions of Notations (CDN) framework [4] to the MiniJava block-based editor. Note that while Holwerda and Hermans focused on the user interface, our focus lies on how the underlying language is mapped to blocks.

Special-purpose Grammar Rules. The MiniJava grammar contains an `ExpressionList` rule for method call arguments:

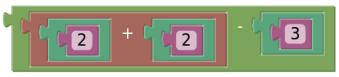
```
syntax MethodCall = Expr "." Identifier "(" ExpressionList? ")";
syntax ExpressionList = Expr ( "," ExpressionList )?;
```

As such, when users want to invoke a method in the derived block-based editor, they experience *repetition viscosity*, where a single desired action in the user's mental model requires multiple repetitive actions, as they have to fetch both a method call block and an expression list block. In the following screenshot, the block equivalent of the expression `this.init(false)` is shown, where the purple expression list block had to be added before the argument could be placed.



Further, these special-purpose blocks will likely contradict the program structure in which users typically think, leading to higher *diffuseness* of the notation, where more space in the notation is taken up to express a certain construct. As the generated editor also does not communicate the need for the expression list block, higher *error-proneness* in the use of the editor can be expected.

Blocks for Leaf Nodes. When writing a simple expression such as `2 + 2 - 3`, users must place a block for each language construct, in this case three numbers and two operators.



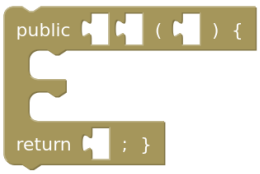
This might make the typical use of numbers and identifiers with operators feel cumbersome. Again, this may lead to *diffuseness* and *viscosity* for entering mathematical expressions.

Similarly, the deeply nested blocks may impact *visibility*, where parts of the notation may not be readily identifiable by the user. Further, users may find themselves routinely pre-fetching number blocks that are no longer needed later on, leading to cases of *premature commitment*. This issue concerns all blocks that will contain leaf nodes in editors generated by Kogi. We also notice the frequent occurrence of special-purpose blocks around leaf nodes, as seen in the above example where the numbers must be doubly nested, requiring three additional blocks.

Limited Reuse of Blocks. Many generated blocks tend to be closely related to one another, such as binary operators, where the structure of inputs stays the same and only a label changes. However, in the MiniJava grammar (see Appendix A), each mathematical binary operator receives its own block type. Consequently, changing an addition to a subtraction operation requires replacing the block, migrating its arguments, and deleting the old operator.

As such, reformulating expressions, even to very similar structures, has high *knock-on viscosity*, where a single desired change cascades to require multiple steps. Further, as users experiment with expressions, they may again have to *prematurely commit* to an operator while still unsure of the algorithm, with high costs to change the operator later on.

Unclear Block Composition. Value and statement inputs in generated editors are not explicitly labeled. Thus users are left to infer the correct types of blocks to place in the open slots based on the knowledge of the underlying language, or worse, the internal structure of the grammar. For example, a method declaration in the MiniJava grammar requires several inputs (e.g., parameters types and identifiers).

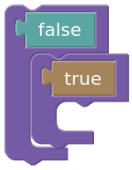


Ambiguous inputs lead to blocks with low *role-expressiveness*, where the purpose of an aspect of the notation is not readily recognizable, and low *visibility* when interacting with the block palette, particularly for novice and end-users.

List Composition with Recursion. Grammars commonly make use of recursion, for example, to define sequencing (lists). In MiniJava, the expression list is defined as:

```
syntax ExpressionList = Expression ( "," ExpressionList )?;
```

As such, users are required to fetch expression list blocks each time they want to add another expression as shown below.



Recursive rules that signify lists have low *role-expressiveness*, as well as low *consistency*, where users apply knowledge from other parts of the notation to new parts, as other forms of lists in the generated editor make use of statement inputs instead. Further, as adding elements requires fetching an additional block each time, the operation has high *viscosity*.

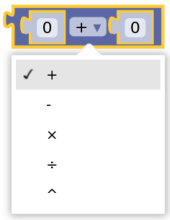
2.3 Aesthetic Criteria

In the following, we briefly describe aesthetic criteria derived from popular, hand-crafted block-based editors, as a set of guidelines for evaluating the improvements of our approach.

Provide Only High-level Constructs. The generated block-based editor should merge special-purpose blocks with the blocks that the user would know based on the language's domain. In the previous example, the method call would thus directly allow placing expression blocks as arguments.

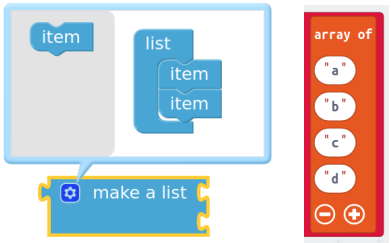
Provide Prefilled Leaf-nodes. Blockly editors can make use of so-called *shadow blocks*, which are placeholders that users can choose to replace or just use as-is. If the editor provides suitable shadow blocks, users can typically use larger block groups directly, without the need to fetch a block for each leaf-node input. In Blockly's example language, the previously shown mathematical expression can be entered using just two operator blocks, which both come with numbers prefilled as their shadow blocks.

Enable Block Reconfiguration. Blockly editors often provide drop-down fields for configuring the exact semantics of structurally identical blocks. For example, the mathematical binary operator in Blockly's example language can change its operator, allowing users to keep the blocks' structure.



Advertise Block Types. Similar to pre-filling leaf nodes, shadow blocks can be used to advertise the type of blocks that can be used in a slot. This additional cue allows users to either recognize the right type of block or find the right type in the palette. In the below example from Blockly, users are shown that the `is empty` block expects a string as input, as indicated by the quotation marks.

Map Lists to Mutators or Statements. Block-based editors using Blockly will typically either use sequences of statement blocks or mutators for lists. Below, on the left, a mutator dialog from MIT App Inventor is shown, where users can add item blocks, thus extending the block on the canvas to take more inputs. On the right, with Microsoft MakeCode's array blocks, inputs can be added using the plus and minus buttons.



3 Grammar Simplification

This section describes our approach to analyze rules within grammars and apply transformations that simplify them, resulting in a block-based editor that follows our previously defined aesthetic criteria. We then outline the changes we made to Kogi's block generation process to further support the aesthetic criteria.

When considering block-based interfaces, Holwerda and Hermans [15] formulate a distinction between language and editor design. During the transformation process, it is important to note that neither the semantics of the language should change, nor should aspects of the language design that are not part of the editor design be changed. Otherwise, if language-specific elements (e.g., labels) are manipulated,

users might not recognize language constructs they are interacting with.

3.1 Simplification Rules

Our simplification pipeline consists of four major phases, which can be further divided into transformation steps. (1) Remove grammatical noise related to encoding operator precedence, (2) remove unnecessary syntax, (3) merge rules to produce a more concise block-based editing interface, and (4) translate the simplified grammar to running code for a block-based editor. On an algorithmic level, most steps work the same: we do a deep pre-order traversal of the grammar's syntax tree and try to match each node's structure to the pattern we want to transform. If a match occurs, we mutate the syntax tree accordingly, either by changing values in existing nodes or by replacing nodes with new ones.

Phase 1: Eliminating Operator Precedence Encoding. The first phase is optional and adapts grammars that have been written in a formalism that does not have explicit support for operator precedence. Without explicit operator precedence, the typical pattern for expressing precedence is via chained, recursive rules, for example:

```
syntax Exp = Exp "+" MulExp | MulExp;
syntax MulExp = MulExp "*" PrimaryExp | PrimaryExp;
syntax PrimaryExp = "(" Exp ")" | digit+;
```

This would lead to different categories of blocks for every level in the precedence hierarchy. Since operator precedence essentially is a disambiguation technique (and such ambiguities cannot exist in a block-based editor), we can safely "squash" such a hierarchy of non-terminals into a single non-terminal, like this:

```
syntax Exp = Exp "+" Exp | Exp "*" Exp; | "(" Exp ")" | digit+;
```

Here, we flattened the chained rules such that it is immediately obvious that each operator's operand must be an `Exp` block. This transformation is applicable, for instance, to left-recursive LR(k) grammars that do not use operator precedence (for instance as provided by Yacc [17]).

Grammars written in formalisms that do not support left-recursion (e.g., standard PEG [11]), have to circumvent it using another common grammar idiom:

```
syntax Exp = Exp1 ("||" Exp1)*;
syntax Exp1 = Exp2 ("&&" Exp2)*;
syntax Exp2 = Exp3 ("|" Exp3)*;
...
```

Once a rule conforming to either of these patterns is identified, we try and locate the top-most rule that no longer conforms to the pattern. Once this rule is identified, we can use it for the left- and right-hand-side operands and transform and inline the derived rules as alternatives of the top-most rule. This type of operation is sometimes known as "deyaccification" [19, 37].

Phase 2: Removing Unnecessary Syntax. In the second phase, we prepare the grammar for further processing by trying to find common patterns of syntactic terminal symbols that are unnecessary in block-based editors. Most importantly, this concerns list separators, as blocks are delineated through the use of user interface elements.

This transformation is realized by searching for commonly used patterns for list separators. For example, the following examples are detected by our heuristic and transformed to the rules with a `_changed` suffix below.

```
syntax List1 = (Element ",")* Element?
syntax List1_changed = Element*
syntax List2 = (Element ",")* Element
syntax List2_changed = Element+
syntax List3 = Element ("," List3)?
syntax List3_changed = Element+
```

Our algorithm traverses the entire grammar once for each type of list structure shown above, matches the expected structure against the current nodes, and, if a match occurs, rewrites the grammar's sub-tree as shown. The transformed rules are then straightforward to translate into *statement inputs* using Kogi's existing transformation logic.

Additionally, in this second phase, we remove all elements in the grammar that only serve to disambiguate the textual sequence of characters and will as such not have an impact on the desired layout and appearance of blocks, such as lookaheads or Rascal's follow conditions.

Phase 3: Merging Rules. In phase three, multiple steps are involved, summarized as follows:

(1) Inline "simple" rules,
(2) merge terminals of structurally identical alternatives,
(3) merge consecutive terminals,
(4) hoist non-top-level alternatives, and
(5) inline chain rules.

Inline "Simple" Rules. By heuristically finding "simple" rules and inlining them, we try to undo the decomposition introduced by the grammar's authors. This step's goal is thus to support our aesthetic criterion of reducing special-purpose blocks. We define a *simple rule* as a rule with the following characteristics: (i) it is not a lexical rule, (ii) it contains at most a single non-terminal (but arbitrary numbers of terminals), (iii) and its top-level expression is not an alternative that includes non-terminals.

Below, we give some examples that match this definition and some that do not:

```
// Does match
syntax Type = "int" | "float" | "double"
syntax Align = ("left" | "right") ("top" | "bottom") Fill?
syntax Group = "(" Expression+ ")"

// Does not match
syntax ComplexType = "int" | "float" | "double" | Identifier
syntax MethodCall = Identifier "(" ExpressionList ")"
lexical identifier = Letter+
```

We do not inline lexical rules because they will be transformed into text fields. We observed that rules with multiple non-terminals are often complex enough; therefore, we found that having a separate block for them was beneficial. Finally, rules that have a top-level alternative, including a single non-terminal, like the `ComplexType` example above (which would match the other two criteria), will become nested alternatives if inlined. However, we want to avoid introducing more nested alternatives, as these will require expanding a block to multiple blocks in the "hoist non-top-level alternatives" step further down the pipeline.

In the concrete example below, we have an expression list rule that no longer contains list separators. Per our definition, it qualifies as a simple rule. When applying the transformation, its usage is replaced by its definition and the `ExpressionList` rule is deleted as it is no longer used in the grammar after inlining.

```
syntax ExpressionList = Expression*
syntax MethodCall = identifier "(" ExpressionList ")"
syntax MethodCall_changed = identifier "(" Expression* ")"
```

This step is applied multiple times throughout the pipeline, as subsequent steps may result in more simple rules to be generated.

Merge Terminals of Structurally Identical Alternatives. This step analyzes all top-level alternatives of a single rule. If it encounters a pattern of more than one alternative that only differ by a single terminal, it will group these. For example, a common case are binary operators:

```
syntax Expr = Expr "+" Expr | Expr "-" Expr | digit+;
syntax Expr_changed = Expr ("+" | "-") Expr | digit+;
```

Here, we grouped all operator terminal symbols in one nested alternative. During the final block generation, this will result in a single binary operator block where all operator symbols are offered in a drop-down list.

This heuristic will ensure that structurally identical blocks end up being convertible in the final block-based editor, thus allowing users to reuse block structures if they only need to change a label. Note that through the previously described "inline simple rules" steps many small differences between rules will already have disappeared. For example, a renaming such as the one below will have been inlined and the alternatives will thus also be considered structurally identical.

```
syntax Expr = Expr "+" Expr | A "-" Expr;
syntax A = Expr;
```

Merge Consecutive Terminals. As a small optimization, we merge consecutive terminals with spaces inserted between them. Otherwise, during block generation, we would generate a separate label for each terminal, leading to large gaps between the words. This step should take place only after matching against structurally identical block, otherwise some previously structurally identical blocks may appear different with merged terminals.

```
syntax Statement = "while" "(" Expr ")" Statement;
syntax Statement_changed = "while (" Expr ")" Statement;
```

Hoist Non-top-level Alternatives. As preparation for generating blocks, we now walk through the entire grammar and locate rules that contain non-top-level, non-terminal alternatives. Any that are found are expanded into separate top-level alternatives. For example:

```
syntax VariableDeclaration
    = (identifier | "int") identifier ";";
syntax VariableDeclaration_changed
    = identifier identifier ";"
    | "int" identifier ";";
```

Without this step, there is no clear mapping to blocks, as alternatives mixing different identifiers or identifiers and terminals cannot be displayed in a field on a block. If there are multiple non-top-level alternatives, we generate the product of all possible combinations.

Inline Chain Rules. If during the above steps any rules ended up being chain rules, we now finally inline these before generating the block-based editor.

```
syntax Function = "function" "(" id* ")" Statement;
syntax Expression = Function | Expression "+" Expression;
syntax Expression_changed = "function (" id* ")" Statement
                          | Expression "+" Expression;
```

If not inlined, the `Function` reference would yield a single empty block with one value input where the dedicated `Function` rule block needs to be inserted to act as an expression. By inlining the `Function` rule instead, a proper `Function` block can directly be used as an expression.

3.2 Block Generation

Once the simplification is done, we take the resulting grammar and generate a Blockly configuration for it. The procedure for this is largely the same as performed by Kogi, except for two important exceptions.

For one, we generate block shadows for each block input, as described in Section 2.3. We do so by trying to determine the most primitive alternative that each rule is offering by looking for blocks with as few inputs as possible but preferring those that contain just a text field. For inputs that require a grammar's "Expression", this will most likely be an identifier or number block. If no good match is found, we pick the first alternative. Even a random pick will, at a minimum, give away the right color of the input block and can be hovered for a textual description, but, more commonly, may also include visual cues such as keywords or other descriptive terminals.

Second, repeated or optional expressions that contain more than just a single identifier are placed in a special mutator inspired by Microsoft MakeCode's implementation with Blockly. An example can be seen in Section 2.3, where the plus button can be used to add further arguments. This

special mutator button is necessary as there is otherwise no mapping of rules such as this one:

```
syntax MethodDeclaration = Type identifier "(" (Type identifier)* ")"
```

To be able to enter the repeating sequence `Type identifier` it would either need to be extracted into a separate block or be added to the parent block as dynamic new inputs. This dynamic type of block is enabled by the mutator. Similarly, optional elements in the grammar can be toggled using the same interface.

4 Implementation

In this section, we present S/Kogi's architecture and selected implementation details. S/Kogi is an extension of the previously described Kogi [33]. Unlike Kogi, which was implemented in the Rascal Language Workbench, S/Kogi is an alternative implementation in Squeak/Smalltalk [16]. It uses a generalized superset of Rascal and Ohm grammars as its input, as special features of neither grammar dialect are required for the simplification or block generation process.

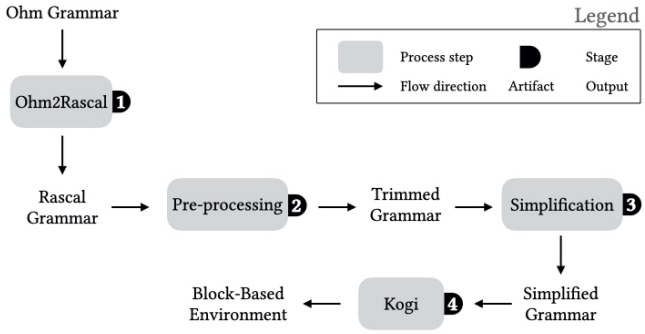


Figure 2. S/Kogi's architecture.

Figure 2 shows S/Kogi's architecture. While no special features from the grammar dialects are required, their structure tends to differ as outlined in Section 3.1. As such, there are two entry points for using S/Kogi: a *Parsing Expression Grammar* (PEG) [12]/Ohm grammar or a CFG. If the input is a PEG, we first transform it into a Rascal-like grammar. Through this step, we reach the second entry point that applies to *Context-Free Grammars* (i.e., Rascal grammars) and now also our modified PEG grammars. The input grammar then traverses the pipeline steps described in Section 3.1. The simplified grammar, conceptually, can then be passed to the original Kogi. In our concrete implementation, the translation step from the simplified grammar to Blockly code for the block-based editor was also modified as described in Section 3.2.

Users can apply minor customizations through the grammar's domain objects in Smalltalk code, as discussed in Section 5 and Section 6. An example of a block generation invocation including customization is shown in Listing 1.

Here, the user renames all usages of the `VariableDeclarationNoIn` rule to omit the `NoIn` suffix and then deletes the rule, before the grammar is passed to the pipeline. After the pipeline has finished, the user assures that `Statement` rules will receive statement outputs instead of value outputs, to override our built-in heuristic. Finally, the user specifies that the optional `Ohm2Rascal` phase should be run and invokes the pipeline, which generates an HTML file and opens it in the user's browser.

```
BlockGenerator new
    grammar: '...';
    preDo: [:g | | oldRule |
        oldRule := grammar ruleNamed: 'VariableDeclarationNoIn'.
        oldRule allUsagesDo: [:identifier |
            identifier contents: 'VariableDeclaration'].
        oldRule delete];
    postDo: [:g | (g ruleNamed: 'Statement') kogiOutput: #statement];
    isOhm: true;
    simplifyAndOpen
```

Listing 1. Customization for the editor generation process.

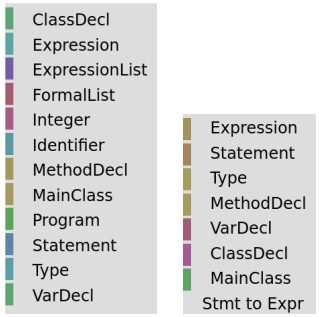
5 Evaluation

In this section, we will first evaluate the impact of our simplification rules on MiniJava before considering several other language grammars with different purposes as small case studies.

5.1 Simplified MiniJava

As previous research [22, 33] had demonstrated limitations of the applicability of generated general-purpose programming (GPL) block-based environments, we will describe the improvements in MiniJava based on our aesthetic criteria defined in Section 2 more closely. As part of the further case studies, we also offer a short evaluation of JavaScript, a complete GPL.

Provide Only High-level Constructs. Below, we show on the left Kogi's original MiniJava palette and on the right, the version generated by S/Kogi.

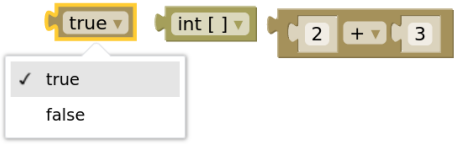


Except for the "Stmt to Expr" category (discussed in Section 6.1), all terms found in the S/Kogi palette should appear familiar to Java developers. Exceptions in the Kogi palette are the `FormalList` and `ExpressionList` rules that are both recognized as "simple" rules in S/Kogi and inlined.

Provide Pre-filled Leaf-nodes. In the below example, we show the variable declaration and binary operator blocks generated by S/Kogi.

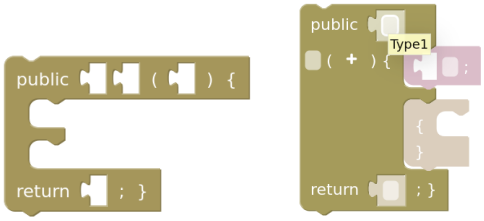
In both cases, our heuristic selected the simplest block of the options (in the first case of rule `Type`, and in the second of rule `Expression`), both containing just a text field.

Enable Block Reconfiguration. Our step of merging terminals of structurally identical alternatives allowed the type and operator blocks to offer drop-downs, rather than appearing as individual blocks each.



As such, users can quickly change between the related instances of Boolean, type, and operator blocks.

Advertise Block Types. The method block in S/Kogi on the right offers users a way to either visually distinguish the types of input blocks, or, if the shapes are ambiguous, to hover blocks and see a tooltip that indicates the block type.



For example, in Kogi's version on the left, users may need to resort to trying various types of blocks to find out that the first statement input is meant for variable declarations and only the second is meant for statements. In S/Kogi's version, the types of blocks are hinted at through the block's shape.

Map Lists to Mutators or Statements. In the above figure, one can see a plus-sign mutator (+) that allows users to add more arguments to the method block. Similarly, repetitions of single identifiers are turned into statement inputs, as typically expected for Blockly-based editors.

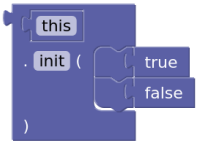


Figure 3. A statement input, allowing users to add multiple blocks without use of recursive blocks.

5.2 Case Studies

In the following, we describe a broader range of grammars and their generated block-based editors. Examples of each can be found in Appendix B.

Cloud Configuration Language (CCL). This language is inspired by the format in Amazon Web Services CloudFormation [31] for allocating cloud resources. It differs from typical DSLs or GPLs in that its structure is fixed and only values change for the most part.

For this grammar, the "inline simple rules" step has the strongest impact. While the block-based editor generated by Kogi places each of the mandatory fields in their own blocks, our simplified version provides one large block, with all mandatory inputs already in place. Similarly, the value inputs are turned into inline input fields and drop-downs.

Questionnaire Language (QL). This language supports the definition of interactive questionnaires and was used to evaluate and benchmark language workbenches [10]. QL targets end-users and supports basic control-flow structures. We reused an existing implementation in Rascal [20].

For QL, merging terminals of structurally identical alternatives was responsible for the most significant cleanup of blocks, as it otherwise comes with 12 separate binary operators. Here too, inlining simple rules ensured that the question block had all its mandatory inputs already built-in.

Sonification Blocks. This language is designed for data sonification and teaches basic principles of sound production, programming, and data flow manipulation [2]. The language's authors heavily customized the block-based editor, for example by adding images of sound waves in blocks.

The grammar [21] contains several alternatives that act as chain rules. These were translated by Kogi to blocks that take a single input of the indicated type, making it exceedingly difficult for users to find the right match. In the simplified version, these alternatives have been inlined, such that users can directly use the high-level block.

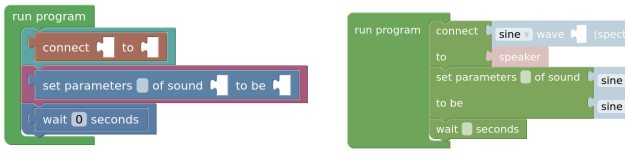


Figure 4. On the left, Kogi's output requires placing distinct wrapper blocks for each type of statement. Our simplified version on the right allows using the statements directly.

State Machine. This language is a simple DSL for describing state machines [33]. For our quantitative evaluation, it acts as a baseline, since it is rather small and the generated block-based editors are essentially identical, demonstrating

that only with a minimum of complexity any benefits of S/Kogi can be observed.

Java. As a popular language [6] and a language of a size that challenges the block-based interaction metaphor we included Java to explore our approach's applicability to complex GPLs.

Interestingly, the S/Kogi version of the grammar [5] yielded significantly more blocks than the Kogi version, as shown in Table 3. This is mostly because lists of expression rules were unrolled and could not be merged in phase 1 of the pipeline because various parts of the grammar referred to subsets of the expression rules. Merging these would have thus created an invalid grammar. When S/Kogi then started inlining rules, it ended up creating duplicates of the expression rules in various places of the grammar. The total number of categories, mapping to groups of language constructs, was still halved in size by S/Kogi, however.

For this type of grammar, it may be desirable to create an invalid grammar that where making more aggressive simplifications is allowed but generate code for a linter that can guide users if they attempt to combine blocks that are not legal in Java. Further, some of the restrictions are not relevant in a block-based editor, for example, several rules have duplicate versions that explicitly exclude "short if statements" as these would clash syntactically.

JavaScript. We used a version of JavaScript with some extensions on top of ECMAScript 5, used for teaching source-to-source transformations (desugaring) using Rascal [8], for which a Rascal grammar was available. Its grammar is similar to MiniJava's but offers more language constructs and presents different structures of rules.

Because of the size of the grammar, using the block-based editor generated by Kogi is difficult: many blocks that need to be combined to form a semantic unit are spread across various categories in the palette. Further, the type of block needed in a given slot is not always clear. Our simplified version improves the generated block-based editor by, first, aggressively merging and inlining rules to end up with a select few that indeed roughly correspond to what a JavaScript programmer may name as language constructs in the language, and second, through the use of shadow blocks that hint the types of blocks that are required as inputs.

Some peculiarities of JavaScript are still found in our simplified version, however: for example, there is a separate block for property assignments because in JavaScript these can be either strings, numbers, or identifiers. As such, property assignments do not qualify as simple blocks and will thus not be inlined. To remedy this, our logic for detecting appropriate shadow blocks will select the likely most commonly used identifier by default for object properties. Thus, users will rarely have to interact with the other property assignment blocks and can use the inlined shadow block directly instead.

Simple customizations can improve some issues. For example, JavaScript contains two variable declaration rules:

```
syntax VariableDeclaration = Id "=" Expression | Id;
syntax VariableDeclarationNoIn = Id "=" Expression!in | Id;
```

The second declaration excludes the JavaScript "in" expression from appearing and is used in some contexts where the "in" expressions are not valid in the grammar, which is only important for correct parsing. In Listing 1 we show a customization users can apply to the grammar to merge the two separate rules, as doing this in an automated manner will likely be prone to produce false positives.

5.3 Complexity Reduction

This section presents a quantitative evaluation of the results obtained after generating block-based environments using both Kogi and S/Kogi for each of the six case studies described earlier. First, we look at the number of Source Lines of Code (SLOC) for each generated environment as a proxy for editor complexity. To measure the SLOC of each case study, we used SonarQube [30]. Then, we look in more detail at each of the generated editors to evaluate the number of categories in their palettes and the number of blocks of each language in both (Kogi and S/Kogi) versions.

Table 2 presents a detailed view of the number of SLOC per environment for each case study. The first column contains the name of the language. The following two columns contain the information regarding the environments generated using Kogi and S/Kogi, respectively. Each of these columns is divided into two sub-columns that contain the number of generated XML and JS (JavaScript) SLOCs. The number of HTML SLOCs is not included in the table because all the case studies were embedded into the same HTML application containing 23 SLOCs. In most environments generated using S/Kogi, there is a reduction in the number of SLOCs, except for the *State Machine* language in which there is an increase of almost 10% in the number of JS SLOC. The reason for this is that S/Kogi uses additional Blockly features (e.g., shadow blocks). In the remaining case studies, there was a reduction in the number of SLOC, and the most significant impact is evidenced in the *JavaScript* language, with a reduction of more than 86% SLOC compared to the Kogi version. The reduction in the SLOC in most of the case studies might benefit language engineers and developers for further fine-tuning their environments since the projects are smaller and, therefore, may be easier to modify than projects with more SLOC.

Table 3 displays descriptive statistics about the block-based editors generated for each of the case studies. The table is divided into two columns, *Kogi* and *S/Kogi*. The first contains the information related to the block-based environments generated using Kogi, and the latter contains the information of the environments generated using S/Kogi. The table shows three main aspects of the generated block-based

Table 2. Comparison between the number of lines of code (SLOC) of block-based environments generated by Kogi against S/Kogi.

Languages	Kogi (SLOC)		S/Kogi (SLOC)	
	XML	JS	XML	JS
State Machine	15	142	19	155
MiniJava	63	1217	147	320
CCL	41	401	8	146
Sonification	71	926	94	263
QL	37	797	66	246
JavaScript	231	4580	341	598
Java	1022	13336	3803	5381

editors; it counts the number of categories in the palette of each language (*# Blocks*), the total number of blocks per language (*# Cats.*), and the number of blocks required for defining an example program in each environment shown in Appendix B.

Based on the collected results, we observe that through the simplification rules, environments generated using S/Kogi have fewer blocks, which is an expected result since that was one of the limitations that we identified in Section 2. The case study that presented the most significant reduction in the number of blocks is *CCL*; the S/Kogi version has almost 88% (14) fewer blocks than the same environment using Kogi. The only exception is the Java language, for the reasons described in Section 5.2. Looking at the number of categories, on the one hand, the language that benefited the most with fewer categories is also CCL with more than 81% (nine categories) fewer categories. On the other hand, the *State Machine* and the *QL* languages were the ones whose palette was reduced but less than the other case studies with only 25% (one category) fewer categories. The reduced number of blocks and categories does not mean that the editors generated by S/Kogi are less expressive, but they inline rules based on the heuristics defined in Section 3. Overall, as programs written using editors generated by S/Kogi require fewer blocks, as shown in *# Block Prog.* columns (Table 3), it is expected that users will find them easier to use. A discussion of this follows in Section 6.2. Similarly, as palettes contain fewer categories and blocks, users' cognitive load is likely reduced when looking for a specific construct or browsing the available blocks.

6 Discussion and Future Work

As described, S/Kogi offers improvements with respect to our established aesthetic criteria over Kogi. This section discusses consequences and limitations of the proposed simplification rules and points out directions for future work.

6.1 Statement vs. Expression Ambiguity

Blockly requires blocks to either occur as values or as statements, each with a differing jigsaw puzzle socket. An example of both types can be seen in Table 1. If a block occurs in both contexts, we provide a rubber element as a workaround, seen for example in Figure 3. However, since the rubber element always requires users to think about the context they want to use a block in, the default shape of a block should reflect its most common usage.

To find this usage, we currently defer to a user annotation in difficult cases. For instance, the example below may lead to false assumptions about the default intended shape:

```
Statement = "if" Expression Statement
          | "{" Statement* "}"
          | ...
```

Here, statements are used as a simple non-terminal and a form in curly braces is provided to parse a sequence of multiple statements. Similar patterns are commonly found for expressions, so a heuristic based on this pattern is not feasible:

```
Expression = "[" Expression* "]" | ...
```

The built-in heuristic counts occurrences of a rule in repeating contexts vs. non-repeating contexts and chooses the more common option. If this choice contradicts the intended semantics, user intervention through an explicit tag is required:

```
(grammar ruleAt: 'Statement') kogiOutput: #statement
```

Here, we get the "Statement" rule and set its output to explicitly be of statement-type for Blockly, such that the generated block can be directly repeated.

6.2 Inlining Depth

Our current approach focuses on reducing the number of blocks as much as possible. At times, it may be desirable for blocks to remain separate. For example, rather than combining all binary operators into one block, it may benefit users to have all arithmetic operators in one block, and all comparison operators in a separate one.

Further, fewer, more complex blocks may sometimes also hinder usability: similar to how users may copy-paste only a region of a larger construct like a method declaration in text, duplicating a part of a more complex block may sometimes be desirable. An example may be found in the Cloud Configuration Language, where almost the entire language is reduced to a single block through our simplifications. While this makes creating a single instance quick and easy, copying just a part of the configuration to another instance becomes significantly more difficult as decomposing and copying just parts is no longer possible with the combined block.

On this end, we considered allowing users to configure how often the inlining rules are being called. This would

Table 3. Comparison of the total number of blocks, palette categories, and blocks used in an example program in block-based editors generated by Kogi and S/Kogi, and the reduction from Kogi to S/Kogi. Note that for Java derived types in the block-based editors were broken, not allowing to create a full program.

Languages	# Blocks			# Cats.			# Blocks Program		
	# Kogi	# S/Kogi	Reduction	# Kogi	# S/Kogi	Reduction	# Kogi	# S/Kogi	Reduction
State Machine	4	3	25%	4	3	25%	8	6	25%
MiniJava	36	25	30%	12	8	33%	47	17	34%
CCL	16	2	88%	11	2	82%	15	3	80%
Sonification	38	18	53%	15	7	53%	13	6	54%
QL	26	14	46%	4	3	25%	21	9	57%
JavaScript	152	63	59%	38	9	76%	36	15	58%
Java	507	664	-19%	256	135	47%	N/A	N/A	N/A

Figure 5. On the left, only shadow blocks are used. The expression can only be a number unless users fetch their own assignment block first. On the right, the assignment is already added, but the expression is left as a shadow, allowing the "12" to be replaced by other blocks.

allow users to control how many levels deep the inlining process should go. Additionally, we considered allowing users to *pin* rules, thus telling the system to not further inline a specific rule.

6.3 Block Shadows

As demonstrated in our examples, Blockly supports a concept of block shadows, where a shadow, for one, signals the type of block that fills an input, and second, if chosen well, allows users to directly use the shadow rather than having to fetch a block for a leaf node. Our heuristic for identifying appropriate default shadow blocks worked out well for our examples, but already, if the rules are specified in different orders, the heuristic could make sub-optimal choices. Additionally, it is not only possible to combine a block with shadow blocks, but also pre-build larger constructs of multiple blocks that users fetch from the palette all at once, as seen in Figure 5. We allow users to specify that this is desirable for certain rules but currently make no attempt to compute this ourselves.

6.4 Block Labels

S/Kogi currently does not generate optimal layouts of labels on blocks. We considered remembering the boundaries of inlined rules and using these boundaries as markers for when a line break could be appropriate. Even then it remains questionable whether labels will end up optimal, without involving the user to manually set line breaks and spaces.

S/Kogi only removes terminals that it recognizes as list separators. As other delimiters such as curly braces are not needed as syntactic dividers in a block-based editor, we considered also removing these but found that in many cases this would lead to seemingly identical blocks. For example, in JavaScript, there is a block for an array surrounded by square braces, and a block for a sequence of statements surrounded by curly braces. These act as important signifiers for users to identify the type of block, in particular, if they have prior experience with the underlying textual language.

The name of the rule of the grammar is already used as the tooltip for blocks. To help disambiguation, S/Kogi could try and detect cases where blocks appear ambiguous and insert the name of the rule as a label on the block.

6.5 Lexical Rules

The handling of lexical rules, summarized here as rules that would likely yield a single token in a traditional parser such as a single literal, is not ideal in S/Kogi. At the moment, we simply always generate a single block with a text field for each lexical rule. This has some benefits and some downsides: for one, there is no distinction between numbers or identifiers, so users are free to type either, and during export, a distinction has to be made involving the help of a parser. If there was a distinction, we could also make better use of the various types of fields that Blockly offers for different input data. Block-based editors also commonly make use of domain-specific graphical elements for data entry, so S/Kogi could allow users to customize the appearance of lexicals in the future.

6.6 DSLs vs General-Purpose PLs

Our case studies illustrate that once a minimum level of complexity is exceeded, S/Kogi will significantly reduce the number of blocks in block-based editors compared to Kogi.

We argue that for language grammars with significant complexity, such as Java, where S/Kogi performs worse, it may be infeasible to create suitable block-based editors, using Blockly's patterns. In Java, many optional special cases exist

throughout the grammar. When writing Java textually, users can omit optional elements such as annotations, while in a block editor, explicit actions for creating optional elements must exist, for example through mutators.

Additionally, especially for larger languages and sometimes in our smaller examples, different groupings for palettes could have been helpful to the user. For example, while the `switch` part of a switch-case statement is placed in the statement category in JavaScript, the `case` element is placed in the separate CaseClause category.

6.7 Usability

As shown in Section 5, the block-based environments generated using S/Kogi contain fewer blocks and categories. This is translated to less visual noise for users; the search space for browsing language constructs is smaller than in the same environments generated using Kogi. As shown in the case studies, users need fewer blocks to create their programs. While creating the example programs, we noticed that the time required to define programs is shorter in S/Kogi generated editors. However, we plan to conduct a formal user study to determine whether this is true for other programs and languages and how time relates to the user's experience.

7 Related Work

There is a lack of tools that help users to develop block-based environments [24]. Most of the existing tooling requires developers to make manual implementations of the desired block-based environments. *Programming environment generation* is an active research line focused on developing tools for existing and new languages. In this direction, *Rascal2MPS* [22] follows a similar approach as the one used in *Kogi* [23, 33]; it analyzes CFGs to derive projectional editors. However, Rascal2MPS presents some limitations, as described by the authors, regarding the usability of the generated projectional editors. S/Kogi is a first step towards generating better editors by analyzing language definitions. For instance, using Blockly's mutator features improves the creation of structural editors, which resembles in a way the so-called editor actions of MPS [26], or transformations in the Synthesizer Generator [29]. Some of the simplification rules described in Section 3 might provide good results for improving the generated projectional editors as well.

The transformations described in this paper can be seen in the context of *grammar convergence* [19]. The goal of grammar convergence is to align grammatical structures represented in different formalisms or styles, and to establish equivalence properties. As an example, consider two grammars for the same language, written for different parser generators, such as ANTLR and Yacc. Both parser generators use different algorithms, and hence require different idioms to encode certain syntactic structures or disambiguation. Examples of such idioms are left-recursion removal, or encoding precedence using a hierarchy of non-terminals. Convergence then consists of systematically transforming one grammar to the other. Examples of such transformations include deyaccification (if a grammar formalism does not support explicit precedence handling), left recursion introduction, renaming, etc.

In this paper, existing grammars are taken as input and are "converged", so to speak, to block-based definitions that do not yet exist but follow certain aesthetic principles. Nevertheless, just like in the original work on grammar convergence, equivalence properties are at stake. The generated block-based language should, for instance, allow the construction of all the programs that the original grammar captures. Similarly, the transformations should not introduce ambiguities that do not exist in the original language.

A more specific instance of convergence is abstract syntax generation from context-free grammars [36]. This work was further refined in the SDF syntax definition formalism [14], and later in Stratego/XT where algebraic signatures are derived from context-free grammars [9]. In a sense, the block-based definitions derived from grammars in this paper are a specific kind of abstract syntax, where some details of the concrete syntax are indeed elided (e.g., whitespace, operator precedence, etc.), but others are not (e.g., keyword literals).

8 Conclusion

We described S/Kogi, an improvement over Kogi, to simplify and optimize block-based editors generated using language grammars, such that existing language infrastructure can also be applied to artifacts from the block-based editor. We demonstrated that the simplifications we apply to the grammar significantly reduce the number of blocks in the block-based editor and improve their usability. As evaluation, we showed that languages of different complexities benefit from the simplification process and only rarely small user interventions were needed to arrive at editors that fulfill our established aesthetic criteria. We thus consider S/Kogi an important step to making use of automatically derived block-based editors feasible.

Acknowledgments

This work is supported by the HPI Research School for Service-oriented Systems Engineering[2] and the Hasso Plattner Design Thinking Research Program[3].

A Complete MiniJava Grammar

```
lexical Integer = natural: [0-9]+;
lexical Identifier = id: [a-zA-Z]+;

syntax Expression
       = i: Integer inte
```

[2]https://hpi.de/en/research/research-school.html
[3]https://hpi.de/en/dtrp/

```
        | len: Expression ex "." "length"
        | t: "true" t
        | f: "false" f
        | id: Identifier i
        | this: "this" this
        | ne: "new" "int" "[" Expression a "]"
        | ne2: "new" Identifier b "(" ")"
        | bracket "(" Expression x ")"
        | a: Expression "[" Expression v "]"
        | v: Expression a "." Identifier v
            "(" ExpressionList? opt ")"
        > d: "!" Expression e
        > mul: Expression e "*" Expression e
        > b: Expression e "+" Expression e
              | c:Expression e "-" Expression e
        > e: Expression e "\<" Expression e
        > and: Expression e "&&" Expression e;

syntax ExpressionList
       = e: Expression ExpressionList*;

syntax Statement
       = s1: "{" Statement* body "}"
       | s2: "if" "(" Expression e ")"
              Statement s "else" Statement s
       | s3:"while" "(" Expression e")" Statement s
       | s4:"System" "." "out" "." "println"
              "(" Expression e ")" ";"
       | s5: Identifier i "=" Expression e ";"
       | s6: Identifier i "[" Expression e "]"
              "=" Expression e ";"
       | s67: Expression ";";

syntax FormalList = lst: Type t Identifier i ;

syntax Type = i: "int" "[" "]" | b: "boolean"
          | i2: "int" | id: Identifier i;

syntax MethodDecl
    = "public" Type Identifier
        "(" FormalList v")" "{"
            VarDecl* a Statement* s "return" Expression ";"
        "}";
syntax VarDecl = var: Type Identifier ";";

syntax ClassDecl = clsdcl: "class" Identifier "{"
                    VarDecl* dcls MethodDecl* mtds
                   "}";

syntax MainClass = "class" Identifier "{"
     "public" "static" "void" "main"
          "(" "String" "[" "]" Identifier ")" "{"
       Statement
     "}"
    "}";

start syntax Program = pro: MainClass c ClassDecl* cls;
```

Listing 2. Definition of the MiniJava grammar in Rascal.

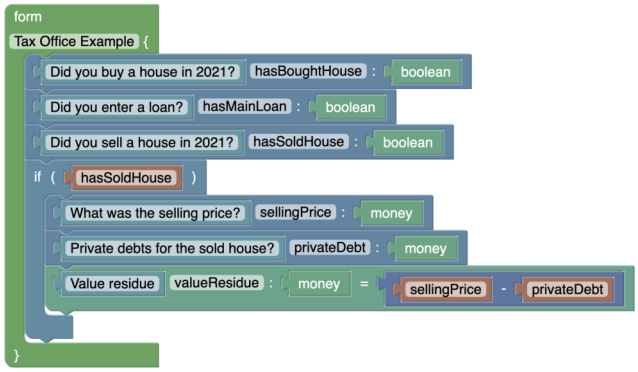


Figure 6. Example program using the QL block-based environment generated by Kogi.

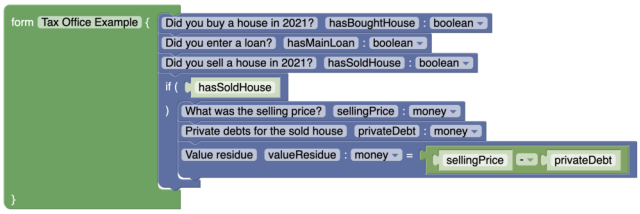


Figure 7. Example program using the QL block-based environment generated by S/Kogi.

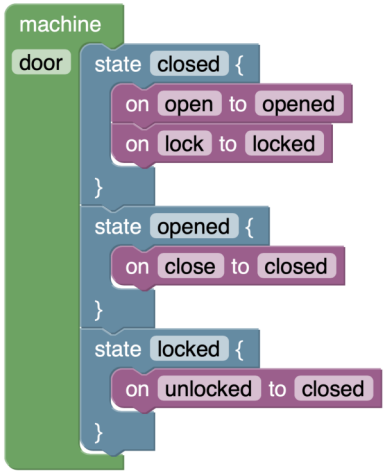


Figure 8. Example program using the State Machine block-based environment generated by Kogi.

B Example Programs

This appendix contain screenshots of the example programs developed in Section 5.

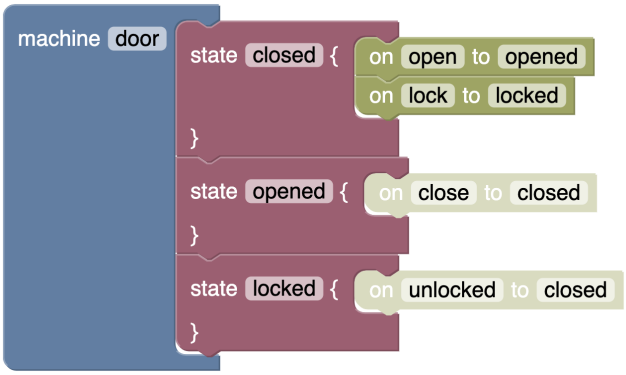


Figure 9. Example program using the State Machine block-based environment generated by S/Kogi. The color of some transitions is different due to the use of the shadow blocks as explained in Section 2.2.

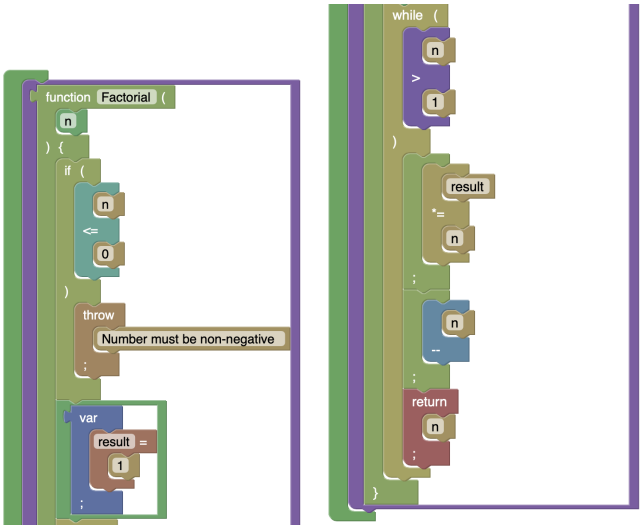


Figure 10. Example program using the JavaScript block-based environment generated by Kogi.

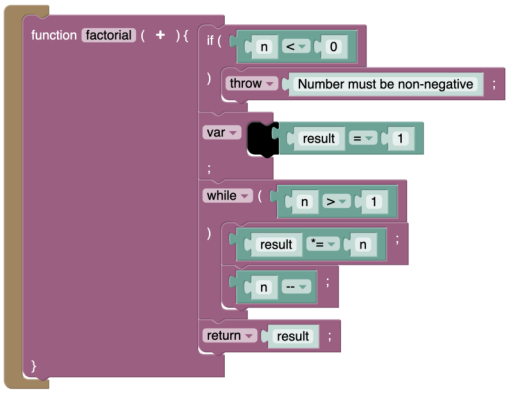


Figure 11. Example program using the JavaScript block-based environment generated by S/Kogi.

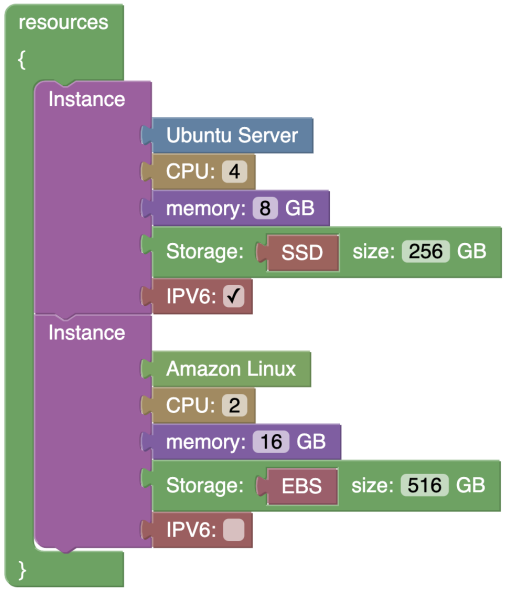


Figure 12. Example program using the CCL block-based environment generated by Kogi.

Figure 13. Example program using the CCL block-based environment generated by S/Kogi.

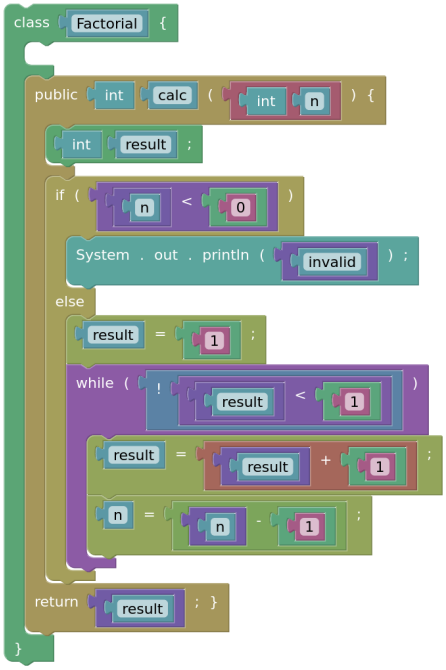


Figure 14. Example program using the MiniJava block-based environment generated by Kogi.

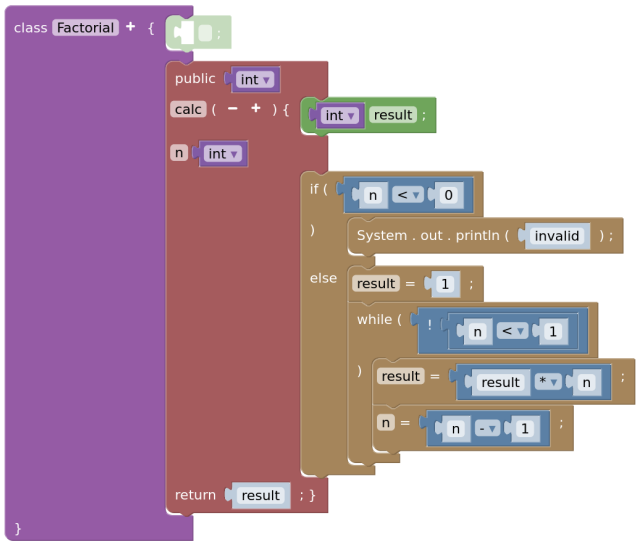


Figure 15. Example program using the MiniJava block-based environment generated by S/Kogi.

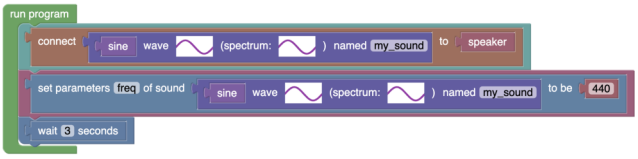


Figure 16. Example program using the Sonification blocks block-based environment generated by Kogi.

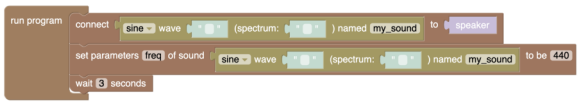


Figure 17. Example program using the Sonification blocks block-based environment generated by S/Kogi.

References

[1] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press.

[2] 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

[3] 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

[4] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind*, Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–341.

[5] Rodrigo Bonifacio. 2008. Rascal-Java8. https://github.com/PAMunb/rascal-Java8/blob/master/src/lang/java/syntax/Java18.rsc. https://github.com/PAMunb/rascal-Java8/blob/master/src/lang/java/syntax/Java18.rsc [Online, accessed 15 July 2021].

[6] TIOBE Software BV. 2021. TIOBE Index for July 2021. https://www.tiobe.com/tiobe-index/. https://www.tiobe.com/tiobe-index/ [Online, accessed 15 July 2021].

[7] João Cangussu, Jens Palsberg, and Vidyut Samanta. 2002. The MiniJava Project. https://www.cambridge.org/us/features/052182060X. https://www.cambridge.org/us/features/052182060X [Online, accessed 12 October 2020].

[8] CWI-SWAT. 2019. SweeterJS. https://github.com/cwi-swat/hack-your-javascript. https://github.com/cwi-swat/hack-your-javascript [Online, accessed 12 July 2021].

[9] Merijn de Jonge, Eelco Visser, and Joost Visser. 2001. XT: a bundle of program transformation tools. *Electron. Notes Theor. Comput. Sci.* 44, 2 (2001), 79–86. https://doi.org/10.1016/S1571-0661(04)80921-6

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël 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 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[11] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (Jan. 2004), 111–122. https://doi.org/10.1145/982962.964011

[12] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[13] Google. 2020. Blockly. https://developers.google.com/blockly. https://developers.google.com/blockly [Online, accessed 13 July 2021].

[14] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. 1989. The syntax definition formalism SDF - reference manual. *ACM SIGPLAN Notices* 24, 11 (1989), 43–75. https://doi.org/10.1145/71605.71607

[15] Robert Holwerda and Felienne Hermans. 2018. A Usability Analysis of Blocks-based Programming Editors using Cognitive Dimensions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 217–225. https://doi.org/10.1109/VLHCC.2018.8506483

[16] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[17] Stephen C. Johnson. 1979. *Yacc: Yet Another Compiler-Compiler.* Technical Report.

[18] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 168–177. https://doi.org/10.1109/SCAM.2009.28

[19] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5423)*, Michael Leuschel and Heike Wehrheim (Eds.). Springer, 246–260. https://doi.org/10.1007/978-3-642-00255-7_17

[20] Mauricio Verano Merino. 2020. *maveme/bacata-demos: First release.* https://doi.org/10.5281/zenodo.3636103

[21] Mauricio Verano Merino. 2020. Rascal - Sonification Blocks. https://github.com/cwi-swat/kogi-examples/blob/master/src/kogi/sonification/Syntax.rsc. https://github.com/cwi-swat/kogi-examples/blob/master/src/kogi/sonification/Syntax.rsc [Online, accessed 12 July 2021].

[22] Mauricio Verano Merino, Jur Bartels, Mark van den Brand, Tijs van der Storm, and Eugen Schindler. 2021. *Projecting Textual Languages.* Springer International Publishing, Cham, 197–225. https://doi.org/10.1007/978-3-030-73758-0_7

[23] Mauricio Verano Merino and Tijs van der Storm. 2020. *cwi-swat/kogi: Kogi 0.1.0.* https://doi.org/10.5281/zenodo.4033220

[24] Mauricio Verano Merino, Jurgen Vinju, and Mark van den Brand. 2021. DRAFT-What you always wanted to know but could not find about block-based environments. (2021). arXiv:2110.03073 [cs.SE] https://arxiv.org/abs/2110.03073 [Under review at ACM Computing Surveys].

[25] 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

[26] JetBrains MPS. 2021. Editor Actions. https://www.jetbrains.com/help/mps/editor-actions.html. https://www.jetbrains.com/help/mps/editor-actions.html [Online, accessed 26 September 2021].

[27] Jakob Nielsen and Rolf Molich. 1990. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. Association for Computing Machinery, New York, NY, USA, 249–256. https://doi.org/10.1145/97243.97281

[28] 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

[29] Thomas W. Reps and Tim" Teitelbaum. 1989. *Defining Hybrid Editors with the Synthesizer Generator.* Springer New York, New York, NY, 95–142. https://doi.org/10.1007/978-1-4613-9623-9_6

[30] SonarSource SA. 2008. SonarQube. https://www.sonarqube.org. https://www.sonarqube.org [Online, accessed 15 July 2021].

[31] Amazon Web Services. 2021. AWS CloudFormation Documentation. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ec2-instance.html. [Online, accessed 12 July 2021].

[32] L. Thomas van Binsbergen and Verano Merino. 2020. Rascal-MiniJava. https://github.com/cwi-swat/rascal-minijava. https://github.com/cwi-swat/rascal-minijava [Online, accessed 12 July 2021].

[33] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/3426425.3426948

[34] 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

[35] 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

[36] David S. Wile. 1997. Abstract Syntax from Concrete Syntax. In *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997*, W. Richards Adrion, Alfonso Fuggetta, Richard N. Taylor, and Anthony I. Wasserman (Eds.). ACM, 472–480. https://doi.org/10.1145/253228.253388

[37] Vadim Zaytsev. 2010. Recovery, Convergence and Documentation of Languages. (Oct. 2010). PhD thesis, Vrije Universiteit.