Language Workbench Support for Block-Based DSLs

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

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam
University of Groningen, Groningen
Netherlands
storm@cwi.nl

Abstract

Block-based languages offer notable advantages for bringing domain-specific languages (DSLs) closer to an end-user programming audience. Nevertheless, the construction of block-based languages is still a rather ad hoc and low-level endeavour. Language workbenches [1] have been shown to be effective in improving productivity when developing textual or otherwise graphical DSLs. In this paper, we sketch open challenges and work in progress to provide language workbench support for block-based languages. In particular we address dedicated meta languages for defining the syntax of block-based languages. Making block-based language development part of the common language workbench repertoire will improve the adoption of the block metaphor outside the realm of programming education, and bring DSLs closer to end-user programming.

Keywords Language workbenches, language formalism, block-based grammar

1 Introduction

Nowadays there are various tools and libraries that support the development of block-based programming environments such as Scratch [6], App Inventor [3], Blockly [5], OpenBlocks [7], and Snap! [4]. However, these tools and libraries focus primarily on the user interface part of a language, and hence lack support to define other aspects of a language processor, such as type checkers, code generators, debuggers, etc.

Language workbenches offer comprehensive meta languages and tools to aid the end-to-end development of (textual) software languages, including IDE services such as syntax coloring, reference linking ("jump to definition"), outline views, error marking, debugging, and others [1]. In this position paper we aim to identify the challenges for bringing both of these worlds closer together.

In particular, we raise the following main question: *What new tools, interfaces, APIs, or formalisms would be needed to bring the development of block-based languages up to speed?* In Section 2 we identify concrete questions regarding syntax definition, abstract syntax tree (AST) representation, and static analysis. In particular, we explore how traditional context-free grammars could be used to define block-based languages in Section 3. Next we first introduce a simple running example to illustrate the challenges.

BLOCKS+, November 04, 2018, Boston, MA, USA
2018.

```
start syntax Machine
  = "machine" Id State*;

syntax State
  = "state" Id "{" Trans* "}";

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

lexical Id = [a-zA-Z]+;
```

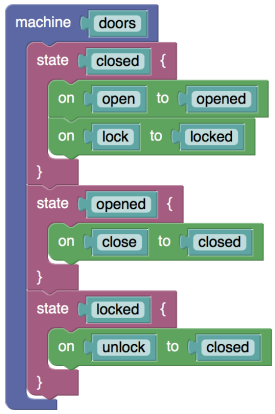

Figure 1. State machine grammar definition and state machine in block notation

Example: State Machines The left of Figure 1 shows the context-free grammar of a simple state machine DSL. This grammar notation is the built-in grammar notation of the Rascal language workbench [2]. A state machine consists of a number named states, which in turn may contain any number of transitions, from a named event to a target state. The right of Figure 1 shows a state machine in block-based notation, modeling the opening, closing, and locking of a door. As can be seen from the figure, there is a close correspondence between the context-free grammar on the one hand, and the structure and labels of the blocks on the other hand. In Section 3, we discuss this relationship in more detail based on our work on mapping arbitrary context-free grammars to Blockly [5] interfaces. First, however, we use the running example to discuss the general challenges of defining block-based languages.

2 Challenges

Syntax Definition Existing systems and libraries for defining block-based languages require either low-level programming or graphical configuration of blocks and their structure. Textual computer languages have since long been described using context-free grammars, which provide a declarative

specification of the syntax of a language. What would a declarative language for block look like? Block-based languages are different from textual languages, in that they define more of the layout (e.g., horizontal vs vertical stacking, left-to-right connecting, or nesting, etc.), often require internationalization, and support multiple widgets for "terminal" symbols (e.g., names, literal values, etc.).

Abstract Syntax Whereas the previous point is concerned about defining the *concrete* syntax of block-based languages, defining the abstract syntax is equally important for implementing language processors. Language workbenches typically employ algebraic data types, or object-oriented class models for this. It is naturally possible to map a block program to some tree structure, but it is unclear, for instance, how much such trees could be typed (e.g., according to the categorization, or allowed nesting of blocks), and how references to outside entities (i.e., entities not literally part of the program, like variables and slider values, etc.) should be represented.

Static Checking In block-based languages, the syntax of a program is correct by construction, but the user may still make type errors and name errors. For instance, defining more than one transition on the same event could be considered erroneous because it introduces non-determinism, and the user should be notified about this. Another error could be to refer to a state that is not (yet) defined as part of the machine.

The definition of such static checking can be defined in terms of the language's abstract syntax, but the reporting of errors needs to be closely integrated with the user interface. In textual languages this is often realized using source locations (e.g., line and column number), but such locations are not easily available for block languages. Such identification of syntax elements is also instrumental for other IDE services, such as hover documentation, jump-to-definition, and visualization of debugging state.

To summarize, we posit that the first challenges to be solved for integrating block-based languages into language workbenches are as follows:

- A declarative formalism to define the concrete and abstract syntax of block-based languages.
- A generic mechanism of identifying syntactic elements for presenting the results of both static and dynamic analyses.

3 Kogi - CFG2Blockly

Kogi is a tool for deriving block-based programming environments from context-free grammars, as defined using the Rascal syntax definition facility. The grammar in Figure 1 is an example of such a grammar, and we expect that Kogi will be able to generate a block-based environment as shown on the right of the figure.

The mapping from grammar to Blockly, however, is not one-to-one. We employ the following heuristics:

- The start symbol (e.g., `Machine`) produces a block without top and bottom connections, which clusters all the possible language constructs.
- Each lexical production produces a block represented as a male jigsaw which allows textual input.
- Each context-free production in the grammar produces one kind of block. The look and behavior of the block depend on the symbols that make up the production.
- A nonterminal symbol (e.g. `Id` in the `State` definition) in a production is mapped to an input block (input value).
- A list of nonterminal symbols is mapped to an input statement, which allows top and bottom connections.
- The order of the literal symbols (e.g. "machine", "state", "", "" etc.) are kept in the same position in the block constructs.

As a result of the derivation process, Kogi is able to generate a block-based programming environment using Blockly as frontend.

Our current focus is how to supplant the information that is in grammar with additional block-specific configuration parameters, to guide horizontal vs vertical stacking, colors, categorization, and ensuring that invalid connections between blocks are not allowed.

4 Conclusion

In this position paper, we have presented some of the open challenges for supporting the definition of block-based languages using language workbenches. We posit that syntax definition and source location identification are key requirements. As a first step, we have presented work in progress on Kogi, a tool that analyzes context-free grammars and produces block-based configurations.

References

[1] Sebastian Erdweg, Tijs van der Storm, Markus Volter, and Laurence Tratt et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[2] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*. 168–177.

[3] MIT. 2012. MIT App Inventor. (2012). Retrieved August 23, 2018 from http://appinventor.mit.edu/

[4] Jens Monig and Brian Harvey. 2018. Snap! (2018). Retrieved August 22, 2018 from https://snap.berkeley.edu

[5] E. Pasternak, R. Fenichel, and A. N. Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop*. 21–24.

[6] Mitchel et al. Resnick. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67.

[7] R. V. 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.