Engineering Language-Parametric End-User Programming Environments for DSLs

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen op
woensdag 6 april 2022 om 11:00 uur

door

MAURICIO VERANO MERINO

geboren te Bogotá, Colombia

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
1^e promotor:	prof.dr. J.J. Vinju
2^e promotor:	prof.dr. M.G.J. van den Brand
Co-promotor:	prof.dr. T. van der Storm (Rijksuniversiteit Groningen)
leden:	prof.dr. J. Gray (University of Alabama)
	dr.ir. F. Hermans (Universiteit Leiden)
	dr. I. Kurtev
	dr. L.J.A.M. Somers

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

ENGINEERING LANGUAGE-PARAMETRIC END-USER PROGRAMMING ENVIRONMENTS FOR DSLS

MAURICIO VERANO MERINO

Faculty of Mathematics and Computer Science
Software Engineering and Technology Group
Eindhoven University of Technology

April 2022

For my parents

— Myriam and Bernardo

Dedicated to the memory of
— María Herlinda Rodríguez.

$1934 - \infty$

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADT   Algebraic Data Type

AST   Abstract Syntax Tree

BNF   Backus–Naur Form

CFG   Context-free Grammar

DSE   Domain-Specific Software Engineering

DSL   Domain-Specific Language

EUD   End-user Development

FODA   Feature-Oriented Domain Analysis

GPL   General Purpose Programming Language

HCI   Human-Computer Interaction

IDE   Integrated Development Environment

LOP   Language-oriented Programming

LSP   Language Server Protocol

LWB   Language Workbench

REPL   Read–Eval–Print Loop

SLR   Systematic Literature Review

SLE   Software Language Engineering

An intellectual says a simple thing in a hard way.
An artist says a hard thing in a simple way.

— Charles Bukowski

ACKNOWLEDGMENTS

As we know, a PhD journey is not easy,
many
      blanks
            need
                  to be
                        filled
                            in,
and in most cases, a big part of the journey's success depends on the supervisors. In my case, I was lucky because I had an excellent team that was always backing me up. Therefore, I would first like to express my
                                        deep
                                      gratitude
to my promotors Jurgen Vinju and Mark van den Brand as well as my co-promotor Tijs van der Storm, whose expertise and support was invaluable to complete this dissertation
            successfully.
Your feedback pushed me to improve and sharpen my research skills and
                            brought my work to a
                                      higher
                                          level.
I enjoyed working and collaborating with you, and I appreciate all the help you gave me, especially during
the
challenging
moments.
   Myriam Merino y Bernardo Verano, sólo tengo palabras de agradecimiento por todo el apoyo que me han brindado durante todo el tiempo.
            Siempre han sido
y serán
                        mi inspiración y
                                            mi mayor motivación.
También quiero agradecerle a Ana Yolanda Durán, Andrea Castañeda, Laura Castañeda y a Edgar Castañeda.

SUMMARY

Human-computer communication can be achieved through different interfaces such as Graphical User Interfaces (GUIs), Tangible User Interfaces (TUIs), command-line interfaces, and programming languages. In this thesis, we used some of these interfaces; however, we focused on programming languages which are artificial languages consisting of instructions written by humans and executed by computers. In order to create these programs, humans use specialized tools called programming environments that offer a set of utilities that ease human-computer communication. When creating programs, users must learn the language's syntax and get acquainted with the programming environment. Unfortunately, programming languages usually offer a single user interface or syntax, which is not ideal considering different types of users with varied backgrounds and expertise will use it. Given the increasing number of people performing any kind of programming activity, it is important to offer different interfaces depending on the programming task and the background of the users. However, from the language engineering point of view, offering multiple user interfaces for the same language is expensive, and if we specifically consider Domain-Specific Languages (DSLs), it is even more expensive given their audience and development teams' size. Therefore, we study how to engineer different user interfaces for DSLs in a practical way.

This thesis presents different mechanisms to engineer different language-parametric programming environments for end-users. These mechanisms rely heavily on reusing existing language components for existing languages or helping language engineers define these interfaces for new languages. We mainly studied four technological spaces, namely, *Grammarware*, *Computational Notebooks*, *Block-based environments*, and *Projectional editors*. We present three different language-parametric interfaces for interacting with DSLs, namely *computational notebooks*, *projectional editors*, and *block-based editors*. These interfaces offer different user experiences and rely upon different technological spaces [1] Different notations are associated with different technological spaces; for instance, grammarware is associated with text files, while block-based environments are associated with Blockly and JavaScript files. Therefore, to provide different notations for their languages, we have to "space travel" so that language engineers can select the most appropriate technological space and interface for their target audience. To support this, we defined grammarware as a common starting point to allow traveling to different technological spaces (e.g., computational notebooks space, projectional editors space, or block-based space). Based on this idea, we developed three tools that

---

[1] A technological space is a shared context that contains a standard body of knowledge, concepts, and hosts different notations [195].

allowed language engineers to generate different interfaces for their DSLs based on a grammar definition of the language. Our results show that it is possible to generate these different user interfaces and decrease the effort required to create these. However, additional research is required to improve the usability of the generated interfaces and make the generation of these interfaces more flexible so that users' data can be used as part of the generated interfaces.

SAMENVATTING

Communicatie tussen mens en computer kan gerealiseerd worden door verschillende interfaces zoals Graphical User Interfaces (GUIs), Tangible User Interfaces (TUIs), command-line interfaces, en programmeertalen. In dit proefschrift hebben we sommige van de eerdergenoemde interfaces gebruikt; maar we hebben ons gericht op programmeertalen, kunstmatige talen bestaande uit door de mens geschreven instructies welke door computers worden uitgevoerd. Om programma's te schrijven worden specialistische programmeeromgevingen gebruikt welke een verzameling aan functionaliteiten aanbieden die communicatie tussen mens en computer eenvoudiger maken. Voor het schrijven van programma's is het nodig om de syntax van de taal eigen te maken en om bekend te worden met de gekozen programmeeromgeving. Maar programmeertalen bieden vaak maar één interface of syntax aan, hetgeen niet ideaal is omdat verschillende type gebruikers, met andere achtergronden en ervaringsniveaus, van een taal gebruik zullen maken. Gegeven dat een toenemend aantal mensen programmeeractiviteiten ondernemen is het belangrijk om verschillende interfaces aan te kunnen bieden, afhankelijk van de uit te voeren programmeertaak of de achtergrond van de gebruiker. Echter is het zo dat, vanuit het perspectief van de taalontwikkelaar, het aanbieden van meerdere interfaces voor dezelfde taal kostbaar is. Dit wordt versterkt wanneer we ons richten op domein-specifieke talen (DSLs), gezien de gebruikersgroepen en de grootte van de ontwikkelteams voor dit soort talen. Het is hierom dat wij onderzoeken hoe verschillende gebruikersinterfaces voor DSLs op een praktische manier gebouwd kunnen worden.

Dit proefschrift beschrijft verschillende mechanismen om taal parametrische programmeeromgevingen te bouwen voor eindgebruikers. Deze mechanismen zijn in sterke mate gebaseerd op het hergebruiken van bestaande taalcomponenten voor bestaande talen en op het ondersteunen van taalontwikkelaars bij het definiëren van interfaces voor nieuwe talen. We hebben met name de volgende vier technologische ruimtes onderzocht: Grammarware, Computational Notebooks, Block-based omgevingen en Projectional editors. We beschrijven drie verschillende taal parametrische interfaces voor het interacteren met DSLs, te weten Computational Notebooks, Projectional editors en Block-based omgevingen. Deze interfaces bieden verschillende gebruikerservaringen

aan en zijn daarvoor afhankelijk van verschillende technologische ruimtes ². Verschillende notaties horen bij verschillende technologische ruimtes; bijvoorbeeld, grammarware wordt geassocieerd met tekstbestanden, terwijl block-based omgevingen worden geassocieerd met Blockly en Javascript bestanden. Daarom, om verschillende notaties voor hun talen aan te bieden, moeten we "ruimtereizenßodat taalontwikkelaars de meest geschikte technologische ruimte en interface kunnen selecteren voor hun doelgroep. Om dit mogelijk te maken hebben we grammarware als een gedeeld startpunt gedefinieerd voor het reizen tussen verschillende technologische ruimtes (zoals de computational notebooks ruimte, de projectional editors ruimte, of de block-based ruimte). Vanuit dit idee hebben we drie tools ontwikkeld die het voor taalontwikkelaars mogelijk maken verschillende interfaces te genereren voor hun DSLs op basis van een definitie van een grammatica voor de taal. Onze resultaten laten zien dat het mogelijk is om deze verschillende gebruikersinterfaces te genereren en zo de ontwikkelingskosten te verminderen. Echter is meer onderzoek noodzakelijk om de bruikbaarheid van de gegenereerde interfaces te verbeteren en om het genereren van deze interfaces flexibeler te maken zodat de data van gebruikers ook binnen de gegenereerde interfaces gebruikt kan worden.

---

2 Een technologische ruimte is een gedeelde context van gestandaardiseerde kennis en concepten en bijbehorende notatie [195].

Part I

PRELUDE

INTRODUCTION

Computers are malleable tools that can perform any type of computation, but their power is limited to only the people who know how to program them. In software engineering, we are constantly trying to raise the levels of abstraction to come closer to the domain people are working in. Still, at the same time, we want to benefit as much as possible from all the power and potential offered by computers. Thus, communication is essential, just like in human interactions. Human interactions require a common language (e.g., Spanish or Dutch) and means (e.g., written or spoken) to send and receive messages from another person. Likewise, communication between machines and people requires common languages, which are *software languages*. Sometimes, software languages are also used as a language between humans, and this communication between humans and machines can occur through different tools (e.g., command lines, files, and interactive systems).

The number of people performing any kind of programming activity has increased. Often their background is heterogeneous, which makes it harder for tool builders to offer a single solution that suits everyone's needs. Some tools are more appropriate for a particular task or group of people than others [114, 118]. Therefore, it is important to offer different tools and interfaces so that users can choose the best tool or interface to achieve their goals. Computer programming is becoming widespread [184], just in the United States of America more than 12 million people say that they do some programming at work, and almost 50 million use spreadsheets and databases [252]. This increasing number of programmers (i.e., novice, end-user, and professional) imposes new challenges and research directions for language engineers and tool developers. The chapters of this thesis present some of these challenges and solutions. Overall, this thesis presents different alternatives for interacting with Domain-Specific Languages (DSLs). Therefore, language engineers and users can choose the right tool for the right job or person [184].

The rest of this chapter is structured as follows: first, it presents some background information introducing the building blocks to understand the rest of this thesis; second, it presents some of the highlights of the research together with challenges identified during its development; third, it presents the origin of the different chapters and the structure of this thesis; and finally, it presents a conclusion of the thesis.

1.1 BACKGROUND

This section introduces the building blocks used in the development of this thesis. It introduces the concepts used within the research field of software language engineering. Moreover, some of these concepts have been also applied to other domains (e.g., end-user development).

Language-oriented Programming (LOP) [110, 374] is a software engineering technique for developing software. This technique relies on creating small languages that are targeted to tackle problems in a specific domain. These small languages are also known as DSLs [88, 235]. LOP splits the software development process into three activities, the design of a DSL, the implementation of different language processors such as code generators, interpreters, or compilers, and the usage of the language. The first activity is focused on understanding the problem the software must tackle and the domain in which the software will be used. Then, a DSL is designed in such a way that it captures the domain's knowledge. This allows language engineers to design the language to use domain terminology instead of generic programming language concepts as General Purpose Programming Languages (GPLs) do. Thus, DSL design provides a design of syntax and semantics of the language. Thanks to this abstraction layer introduced by the DSL, the resulting language is closer to the users, and it is focused on the domain, which often makes users' programs more concise and, therefore, easier to read by domain experts. Since DSLs offer high-level language constructs to address a specific problem in a particular domain, they increase the users' productivity.

In addition to the language's design, the second step is to implement the language using some existing meta-language. It is often done by developing different language components that support the its execution (e.g., code generators, parsers, or compilers). There are tools specialized in this domain of creating languages called *Language Workbenches.* It is essential to notice that the DSL program captures the domain knowledge, and then these language processors are responsible for translating that knowledge into the desired target language (e.g., Java, Python, or Assembler). In this second activity, it is important to mention that there could be several translation layers from the DSL specification until the target language; the essential step here is to retain the semantics of the program across the different translation layers. Finally, the third activity is related to language usage; this activity is essential to determine whether the design and implementation of the language meet the established objectives.

Moreover, the LOP success also requires development of proper tooling for interacting with the resulting DSL. This is an active research field focused on designing and building user-centered tools that help users during their programming activities. Building such tools is an essential aspect because users require intelligent tools that allow them to express the solution of a problem in a friendly manner, which is a challenge in the LOP domain. This is a challenge but also an opportunity because development teams and communities behind DSL development are considerably smaller than the

people behind a GPL. Thus, engineers using LOP require tools that help them reduce the time and costs involved in developing tools. As a result, in the academic community there is an active research line on programming environment generation [54, 68, 87, 98, 99, 148, 267, 287]. The current thesis presents contributions in this research line.

In the remaining of this chapter, we present the common building blocks required to understand the contributions of this thesis. This thesis explores different user interfaces that rely on different technologies. Therefore, as introduced by Kurtev et al. [195], we refer to each technology as a *Technological Space*.

TECHNOLOGICAL SPACES   A Technological Space (TS) is a shared context in which there is a standard body on knowledge, concepts, methods, tools, literature, and required skills to solve problems [195]. For instance, *Objectware*, *Javaware*, *Pythonware*, *Grammarware*, *Modelware*, and *XMLware* are typical examples of TSs. Each of these TS offers various benefits and drawbacks, which depend on the problem domain in which they will be used. Nowadays, developers have different TS for solving problems: they are often faced with choosing which TS is more convenient for the problem they are trying to achieve. Sometimes, the desired solution requires more than a single TS: developers need to travel across different TSs, which might be challenging. This thesis allows developers to use different interfaces that rely upon different TS (*Grammarware*, *Computational Notebooks*, *Block-based environments*, and *Projectional editors*).

DOMAIN-SPECIFIC LANGUAGE (DSL)   As introduced before, a DSL is a small programming language tailored to solving problems in a particular domain. There are two main types of DSLs, *internal* and *external* DSLs. The first one is characterized by reusing the concrete syntax and the parser of a host language (e.g., a library), while the latter requires the definition of a custom syntax, parser, and language processors (e.g., compiler, interpreter, and type checker). Since DSLs focus on solving problems in a particular domain, they often include domain concepts in their concrete syntax, which allow users to express solutions to problems in a more concrete and less verbose manner than a GPL. Moreover, DSLs allow users to focus on the domain's problem and forget about the implementation details, as this is the responsibility of the language engineer that develops the underlying code generator, compiler, or interpreter.

The DSL syntax uses domain concepts based on a domain analysis in which the language will be used, and this allows more people to develop software and better quality in less time than when using GPLs. DSLs offer these benefits because they are easier to use and more expressive than GPLs [235]. As we mentioned before, tooling is essential for the success of DSLs. Developing a programming system for a DSL requires a huge effort [167]. That is why tools and research on programming environment generation are essential.

LANGUAGE WORKBENCHES (LWB)   An essential aspect for the success of LOP techniques relies upon the technologies that support the design, prototyping, and development of software languages [197]; these tools are also known as *language workbenches*. A language workbench [99, 110] is an Integrated Development Environment (IDE) for developing software languages. They offer a set of meta-languages for defining the syntax and semantics of a language and also features for developing IDE services (e.g., syntax highlighters, error marking, and auto-completion) for the language. Although all Language Workbenches (LWBs) help engineers in the language development process, there are mainly three different types of LWBs [62, 99, 197] namely *textual* (e.g., JastAdd [98], Rascal [180] , Spoofax [166], and Xtext [102]), *graphical or modelware* (EMF [60], MetaEdit+ [171] and GME [203]), or *projectional* (e.g., JetBrains MPS [62], the Intentional Domain Workbench [321], and SmartTools [19]). Based on the previous description of technological spaces, we can argue that each type of language workbench is a technological space where their communities have a shared body of knowledge, tools, and common practices.

It is crucial to mention that language engineers must choose from picking up the right type depending on the business requirements. All three types are powerful, and they all have some drawbacks, but depending on the requirements, one might be more appropriate than the other. In this thesis, we explore in detail textual and projectional LWBs; the study of *modelware* is out of the scope of this thesis.

TEXTUAL LANGUAGE WORKBENCH   The main characteristic of textual language workbenches is that they rely upon context-free grammars for the language specification. Thus, language engineers must define the language's formal syntax through a grammar that often follows the Backus–Naur Form (BNF) notation or BNF extensions (e.g., extended BNF and augmented BNF). Then, an essential step within this technological space is the *parsing* technology. Programs are parsed against the specification, and if the program conforms to all the rules described in the formal language definition (grammar), then the parser produces a parse tree. A parse tree contains all the information and relationships between the program and the grammar. Some parsers support ambiguous programs, and in such cases, the result is a forest of parse trees instead of a single tree. In this technological space, it is often common to transform the parse tree into an Abstract Syntax Tree (AST) because it is easier for language processors to work with the program's abstract representation since ASTs do not contain layout information; they contain the essential elements of the program solely. In this thesis, we used as a starting point *Rascal*, which is a grammar-based language workbench.

PROJECTIONAL LANGUAGE WORKBENCH   This type of Language Workbench (LWB) is different from the textual counterparts because, in a projectional LWB, users do not interact with a textual representation of the program but with projections of the ASTs. This means that users are directly manipulating the AST of the program during

the program development. As a result, no parsing nor parsing technology is required. Moreover, these platforms support the definition of several notations (AST projections), which means that they can easily support the mixing of different notations (e.g., textual, graphical, and cell-based) in the same program. In these platforms, it is common to find a textual projection of the AST to offer users a text-like experience when developing software. In this thesis, we explored the use of JetBrains MPS as a projectional LWB. Based on the definition that a projectional LWB allows users to directly manipulate the AST of a program, we can consider that a block-based environment is also a sort of projectional LWB. However, for this thesis, the decision was to present block-based environments later in a separate section, since a deeper study on these platforms (Chapters 6 and 7) was performed; therefore, this decision created an overlap because a block-based editor could be seen as a block-based projection of an AST.

COMPUTATIONAL NOTEBOOKS   Computational notebooks are interactive programming environments that allow users to interleave documentation and executable code in the same document. We can think about them as a contemporary type of literate programming [183].

Essentially notebooks are composed of two parts, the *front-end* and the *back-end*. The first is represented as a cell-based document as shown in Figure 1.1. This type of documents contain three main types of cells, namely *documentation* (top part of Figure 1.1), *input* (Figure 1.1 cells containing the prefix in [n]), and *output* cells (Figure 1.1 cells. Documentation cells are used for writing prose, which is meant to describe the executable code or describe the results of the current notebook. Users use the input cells to write executable code snippets, and as a result of their execution, output cells might be obtained. Likewise, there are mainly two types of output cells, one that renders static content (e.g., text or charts) and interactive output (e.g., interactive visualizations that allow users to manipulate variables, and their change is reflected in the visual representation).

The *back-end* of a notebook relies on a language kernel interface. Depending on the notebook platform, they offer support for different languages. However, most popular general-purpose programming languages are supported by them. These platforms offer APIs to support additional languages. However, using these APIs is a cumbersome task because developers must implement a low-level communication protocol, which might not be significant in terms of Source Lines of Code (SLOC) , but it is difficult to debug.

BLOCK-BASED ENVIRONMENTS   Block-based environments or block-based editors are visual interactive programming environments that allow users to create programs through visual representations of language constructs. These environments have some unique characteristics that distinguish them from other programming environments. First, block-based editors represent language constructs using visual blocks that resemble jigsaw puzzle pieces (see Figure 1.2). So that users can create programs as if
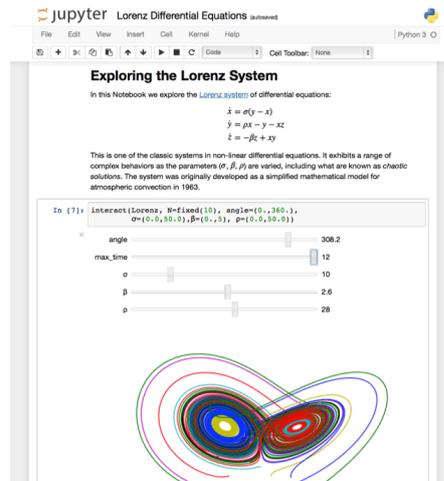
Figure 1.1: Example of a computational notebook document. Image from [161].

they were completing a jigsaw puzzle. Each block (language construct) has a shape and a color. Its shape restricts the options to which the block can be connected. Offering different block shapes helps users identify valid and invalid connections by just looking at their forms.

Block-based environments are often divided in three main parts, namely *palette*, *canvas*, and *stage*. The *palette* (left part of Figure 1.2) contains all the language constructs. It is like a cupboard that stores all language constructs grouped by categories, and these categories are not standard, they change from one language to another. Another essential aspect of a block-based editor is the *canvas* (middle part of Figure 1.2), which is the place in which users develop their programs.

As presented in Section 1.1, textual languages rely on string-based documents to create programs: creating programs is achieved by using the keyboard and writing a sequence of language constructs. Instead, block-based editors rely on a different interaction mechanism, *drag and drop*. With this mechanism, users select a block from the palette, then drag and drop it from the palette into the desired location in the canvas.

Finally, the *stage* (right part of Figure 1.2) displays the output of a program. This component is not standard in block-based environments, but popular block-based editors such as Scratch [240, 288] and Snap! [243] offer one. This space is used to render the results of running the program written on the canvas. Depending on the nature of the language, it can be used to display visualizations, animations, or output text.

Although there are many block-based editors, the number of available tools for developing them is limited. Either developers implement everything from scratch or they rely on these libraries. In this thesis, we used *Google Blockly* [119], which is one of the most popular libraries for developing block-based editors. It allows language engineers to implement block-based editors through a JavaScript API and compile such
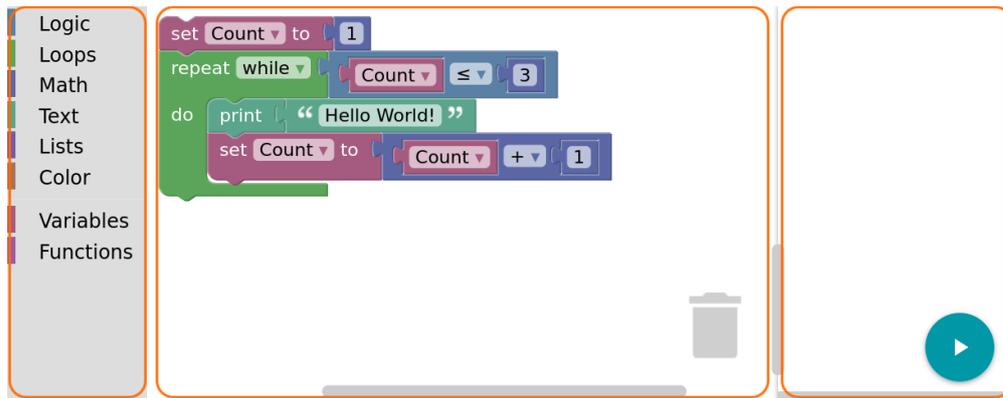
Figure 1.2: Example of a Block-based environment using Google Blockly [119].

programs into a target language (e.g., Python or JavaScript). However, this flexibility means that the translation from the Blockly representation to the target language must be implemented manually by developers.

1.2 RESEARCH QUESTIONS

In the previous section we presented the common knowledge required to understand the rest of this thesis. In this thesis, we studied LOP as a technique for software development that brings several benefits for developers (e.g., user's productivity and usability). This is situated in the domain of software language engineering, and more concretely, in the space of programming environments generation. Offering different user interfaces for the same DSL will provide benefits for different user groups. In this context, the main challenge becomes the design and engineering of these different user interfaces. So we defined a general research question and split it into nine more specific research questions. The general research question is:

> **RQ**: *How to engineer different user interfaces for DSLs, so that language engineers can choose the right technological space and notation for various types of users, while reusing existing language components?*

Addressing this question requires (i) to study different technological spaces, namely computational notebooks, projectional editors, and block-based environments. Based on the above, (ii) we can identify possible alternatives to allow language engineers to travel from one technological space to another, and (iii) evaluate how these alternatives can be used in practice. In particular, we show how to travel from the grammarware space into the other three technological spaces.

Our first research question is based on the manifold benefits that computational notebooks bring when experimenting with code and text (e.g., quick prototyping,

immediate feedback, reproducibility). We noticed that these platforms were available mainly for GPLs. The main reason is that developing tools for languages is expensive: popular GPLs count on big communities that support their development. However, in the case of unpopular languages, these resources are scarce, burdening the development of such tools. Thus, we ask the following research questions:

> **RQ1.1**: *What is required to define a computational notebook at the language abstraction level?*

> **RQ1.2**: *How can notebooks be offered as a generic service in language workbenches?*

Addressing these questions requires understanding what computational notebooks are, their main features, and which programming languages these interfaces support. Based on these abstractions, we can identify and define how to design and implement a language-parametric notebook generator.

Notebooks can be seen as GUIs for command-line interfaces; however, some considerations might differ (e.g., execution order). In this type of interface, users can type valid commands that the underlying compiler or interpreter executes. However, not all language constructs can be used within a command-line interface, only some commands are valid, and they are called *entry points*. This implies some considerations from the language-design perspective so that the syntax of the language and the interpreter (or compiler) support different entry points. In other words, the syntax and interpreter of the language should support the incremental definition of programs. If not, users must write entire programs within a single command, which hampers the usability of this type of interface.

Although most computational notebook platforms offer different mechanisms to support additional programming languages, it is unclear whether there are special requirements for using a language within a notebook interface. With this in mind, the following two research questions arose:

> **RQ2.1**: *What are the requirements for designing a notebook-friendly language?*

> **RQ2.2**: *How can we transform an existing language into a notebook-friendly language?*

Based on our findings from the answers to questions **RQ1.1** and **RQ1.2**, we observed that language design is essential for designing languages that are suitable to be used within a notebook. Therefore, with the previous two research questions (**RQ2.1** and **RQ2.2**), we studied how to identify and design notebook-friendly languages, to open up the notebook metaphor for DSLs to improve the end-user experience when interacting with code and to increase DSLs adoption.

Language engineers can develop tooling for their languages (e.g., notebooks) based on the design decisions of notebook-friendly languages and their implementations. In this way, language engineers can improve the user experience of their languages through a notebook interface. This leads us to ask the following question:

> **RQ3**: *What language design guidelines could be used to improve the programming experience of DSL users within a notebook platform?*

As mentioned before, we explored different TSs, and in the following research question, we studied projectional editors concretely within the domain of JetBrains MPS. Our exploration starts from a textual LWB (Rascal) to enable traveling from the grammarware to the projectional TS. Languages defined using this type of LWB are often described using context-free grammars, while projectional LWB uses other mechanisms. Given the heterogeneity of these TSs, it is necessary to study how these different TSs can be bridged so that language definitions can be reused across them. Therefore, we defined the following research question:

> **RQ4**: *What mechanism can be used to map context-free grammars to projectional language definitions?*

Based on the previous research question results, it is possible to travel to a different TS. So now, we travel to a different TS, block-based environments. To do so, we address the following two research questions.

Block-based environments or editors offer an interesting visual notation for software languages, in which language constructs are represented as jigsaw-like puzzle pieces. Their visual appearance can help users create their programs, as shown in different contexts, such as in education and creative environments for children. However, it is unclear how these programming environments are being used in the wild, how they are being developed, and what are their main components. Moreover, it is uncertain whether these programming environments are being used mainly by children. Therefore, to better understand block-based environments, the following research question was formulated:

> **RQ5**: *What are the main characteristics of block-based environments and how are they implemented?*

After analyzing block-based environments in detail, their characteristics, usage, and development, we want to study and explore if specialized tooling for implementing languages (i.e., language workbenches) can be used to engineer block-based environments in practice. Concretely, we ask:

> **RQ6**: *How can language workbenches support the development of block-based environments by reusing existing language components?*

Using specialized technology for creating and deriving block-based editors is an interesting direction. Previous research has demonstrated that the generation of programming environments increases developers' productivity and speeds up the development process of new tools. However, the resulting environments often require manual intervention from the developers to improve their usability. Therefore, we wanted to explore whether the usability of block-based programming environments can be improved without manual intervention, so we defined the following research question:

> **RQ7**: *What techniques could be applied to improve the usability of generated block-based editors?*

In summary, there are three main concerns derived from our main research question (**RQ**), (i) study of different technological spaces. (ii) identify possible ways of supporting technological space traveling. (iii) evaluate how they can be used in practice. The first one it is discussed in detail through research questions **RQ1.1**, **RQ1.2**, **RQ4**, and **RQ5**; while the second and third concerns are studied from different perspectives and technological spaces through questions **RQ1.1-RQ4**, **RQ6**, and **RQ7**.

The research questions are addressed using different technological spaces: research questions **RQ1.1-RQ3**, are addressed from the *notebook* perspective, **RQ4** is studied from the projectional LWB view, and **RQ5-RQ7** are studied from the block-based perspective.

In the next section, we present how each research question is addressed.

1.3   ORIGIN OF THE CHAPTERS

This section presents the origin of each of the chapters in this thesis and the author's contributions to each of them.

Chapter 2   This chapter is the result of three publications. The main contribution of this chapter is the definition of a Feature-Oriented Domain Analysis (FODA) of computational notebooks. Moreover, it presents a design for creating a parametric language interface to enable communication between notebook platforms and language workbenches by reusing existing language components. Therefore, Bacatá is developed to validate such an interface. Bacatá is a tool that enables DSLs defined within the Rascal LWB to be used as Jupyter Notebooks. Also, Bacatá uses existing language definitions to generate IDE services such as *syntax highlighting* and *auto-completion*. To evaluate Bacatá, we created notebooks for different languages *Calc*, *QL*, *Halide*, and *SweeterJS*. This chapter addresses research questions **RQ1.1** and **RQ1.2**, and it is based on the following publications.

- Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: a generic notebook generator for DSLs." In: Domain-Specific Language Design and Implementation workshop, DSLDI '17. Vancouver, British Columbia, Canada, 2017 [231].

- Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: A Language Parametric Notebook Generator (Tool Demo)." In: SLE 2018 (2018), pp. 210–214. DOI: 10.1145/3276604.3276981 [356].

- Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: Notebooks for DSLs, Almost for Free." In: *The Art, Science, and Engineering of Programming* 4.3 (2020). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/11 [359].

Chapter 3   Based on some of the difficulties that we identified during the development of the papers presented in Chapter 2, we observed that not all languages were 'notebook-friendly'. Thus, to address research questions **RQ2.1** and **RQ2.2**, we present a principle approach for characterizing such languages and their interpreters in this chapter. As a result, we introduce *sequential languages* and how to develop an *exploring interpreter* for such a language. In addition, this chapter presents applications of such interpreters for developing three types of REPL-like systems, namely *notebook*, *command-line interface*, and a *client-server* interface. This chapter is based on the following publication.

- L. Thomas van Binsbergen et al. "A Principled Approach to REPL Interpreters." In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. ACM, 2020, pp. 84–100. ISBN: 9781450381789. DOI: 10.1145/3426428.3426917 [44].

Chapter 4   This chapter presents a possible application of the exploring interpreters in the context of a notebook interface, and it exclusively answers our research question **RQ3**. Mainly, we present how an interactive *execution graph* can be constructed based on the definition of an exploring interpreter and how it can be beneficial for end-users. Likewise, we present how to develop a *variable-watcher* on top of the exact definition of an interpreter.

Chapter 5   This chapter addresses the research question **RQ4** by presenting an approach for deriving projectional editors in MPS from a context-free grammar definition in the Rascal LWB. To achieve this we used heuristics that address the problem of pretty-printing, and we applied them to the context of bridging the gap between projectional and textual languages. As a result, we derive a mapping between a grammar definition into the structure and editor of a projectional language. The content of this chapter is based on the following publication.

- Mauricio Verano Merino et al. "Domain-Specific Languages in Practice with Jetbrains MPS." In: 2021. Chap. Projecting Textual Languages. ISBN: 978-3-030-73758-0. DOI: 10.1007/978-3-030-73758-0 [232].

Chapter 6   This chapter presents a systematic literature review of block-based environments and addresses research question **RQ5**. Based on this study, we identified

that not much attention had been paid to developing these environments from the software language engineering perspective. Proper support for defining and implementing block-based editors might increase and improve the development and adoption of these interfaces, especially for end-user languages (e.g., DSLs). This chapter is based on the following publication.

- Mauricio Verano Merino, Jurgen Vinju, and Mark van den Brand. "What you always wanted to know but could not find about block-based environments." In: (2021). [Under review at ACM Computing Surveys]. URL: https://arxiv.org/abs/2110.03073 [230].

Chapter 7   This chapter addresses the research question **RQ6**. It presents a first step towards adding language workbench support for describing block-based environments. Based on the lack of support for creating block-based editors, we designed and developed a tool that analyzes context-free grammars and produces a block-based environment. The resulting block-based environment can reuse existing language machinery (e.g., type checkers, interpreters, and REPLs). This chapter is based on the following publications.

- Mauricio Verano Merino and Tijs van der Storm. "Language Workbench Support for Block-Based DSLs." In: *BLOCKS+ Proceedings* (2018) [226].

- Mauricio Verano Merino and Tijs van der Storm. "Block-Based Syntax from Context-Free Grammars." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: ACM, 2020, pp. 283–295. ISBN: 9781450381765. DOI: 10.1145/3426425.3426948 [355].

Chapter 8   As a result of Chapter 7, we found that it was possible to derive block-based editors from context-free grammars. However, the resulting editors sometimes contained too many unnecessary blocks, which led to verbose and lengthy programs. In this chapter, we answer research question **RQ7** by defining a pipeline for simplifying the input grammars used to derive the block-based editors. The pipeline contains different phases that transform the original grammar. As a result of applying this pipeline, we show a reduction in the resulting number of blocks, which yields less verbose programs. This chapter is based on the following publication:

- Mauricio Verano Merino et al. "Getting Grammars into Shape for Block-Based Editors." In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2021. Virtual, USA: ACM, 2021. ISBN: 978-1-4503-9111-5/21/10. DOI: 10.1145/3486608.3486908 [360].

Chapter 9   Finally, in this chapter, we revisit all the research questions and summarize the main contributions of this thesis.

1.4 TOOLS AND SOFTWARE

The following tools and languages were designed and implemented during the development of this thesis. Each of these tools is used in one or more chapters, and they are open source and available in their respective repositories.

- Bacatá [357]
- Rascal notebook [229]
- Execution graph [223]

- MiniJava [41]
- Rascal2MPS [27]
- Kogi [227]

- S/Kogi [32]
- PDFMiner [221]
- Halide* [222]

1.5 THESIS STRUCTURE

Figure 1.3 presents the general structure of the chapters of this thesis using block-based notation. This thesis is divided into three main topics. The first one is about computational notebooks and Chapters 2 and 3 addresses research questions **RQ1** and **RQ2**, respectively. The second topic is about projectional editors, and Chapter 5 answers research question **RQ3**. The third topic is about block-based environments, the answers to research questions **RQ4** and **RQ5** are described in Chapters 6 and 7, respectively. Finally, we conclude this thesis by revisiting our research questions and summarizing the main contributions of our work.
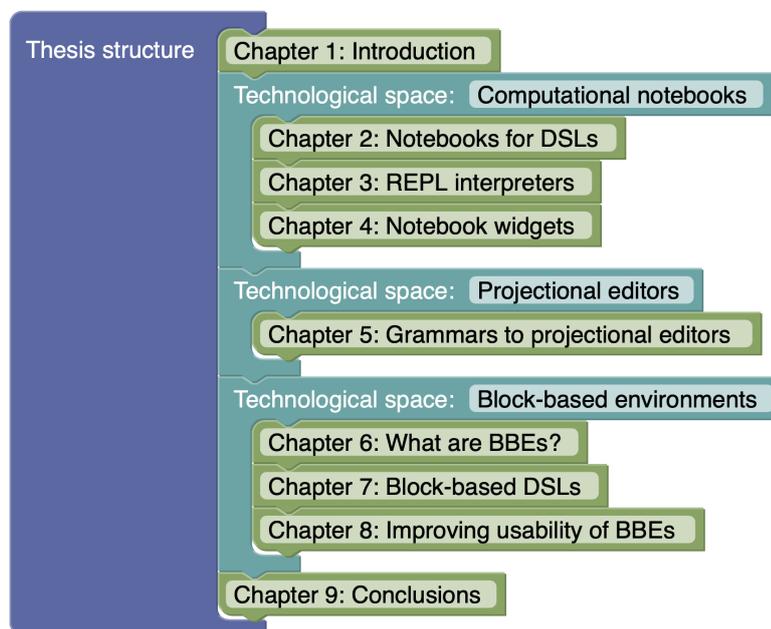
Figure 1.3: Thesis chapter structure.

Part II

COMPUTATIONAL NOTEBOOKS

2

BACATÁ: NOTEBOOKS FOR DSLS, ALMOST FOR FREE

Context: Computational notebooks are a contemporary style of literate programming, in which users can communicate and transfer knowledge by interleaving executable code, output, and prose in a single rich document. A Domain-Specific Language (DSL) is a software language tailored for an application domain. Usually, DSL users are domain experts that may not have a software engineering background. Therefore, they might not be familiar with Integrated Development Environments (IDEs). In brief, the development of tools that offer different interfaces for interacting with a DSL is relevant.

Inquiry: However, DSL designers' resources are limited. We want to leverage General Purpose Programming Languages (GPLs) tooling in the context of DSLs. Computational notebooks are an example of such tools. Then, our main question is: What is an efficient and effective method of designing and implementing notebook interfaces for DSLs? By addressing this question, we might be able to speed up the development of DSL tools, and ease the interaction between end-users and DSLs.

Approach: In this chapter, we present Bacatá, a mechanism for generating notebook interfaces for external DSLs in a language-parametric fashion. This mechanism is designed in a way in which language engineers can reuse as many language components as possible (e.g., language processors, type checkers, code generators). In addition, we present a Feature-Oriented Domain Analysis that depicts language dependent and language independent features of computational notebooks.

Knowledge: Our results show that notebook interfaces generated by Bacatá can be used with little manual configuration. However, there are a few considerations and caveats that should be addressed by language engineers that rely on language design aspects. The creation of a notebook for a DSL with Bacatá becomes a matter of writing the code that wires existing language components in the Rascal language workbench with the Jupyter platform.

Grounding: We evaluate Bacatá by generating functional computational notebook interfaces for three different non-trivial DSLs, namely: a small subset of Halide (a DSL for digital image processing), SweeterJS (an extended version of JavaScript), and QL (a DSL for questionnaires). Additionally, it is relevant to generate notebook implementations rather than implementing them manually. To illustrate this, we measured and compared the number of source lines of code that we reused from existing implementations of those languages.

Importance: The adoption of notebooks by novice-programmers and end-users has made them very popular in several domains such as exploratory programming, data science, data journalism, and machine learning. Why are they popular? In (data) science, it is essential to make results reproducible as well as understandable. However, notebooks are only available for GPLs. This chapter opens the notebook metaphor for DSLs to improve the end-user experience when interacting with code and to increase DSLs adoption.

2.1 INTRODUCTION

Computational notebooks are cell-based documents that allow users to interlace executable source code and interactive computed results with prose that explains them. Wolfram Mathematica is one of the first commercial frameworks for creating notebooks [132, 370]. Lately, notebooks have become popular in disciplines such as mathematics, physics, data science, programming education, and machine learning due to the benefits notebooks provide in terms of reproducibility and usability [25, 85, 153, 259, 271].

Currently, there are several dozens of platforms that support the creation of computational notebooks. Wolfram Mathematica was one of the first platforms for notebooks, yet its appropriation was limited due to their commercial licensing model. Later, in 2014, Project Jupyter [182] developed an open-source notebook platform that has increased the adoption of the notebook metaphor among different disciplines thanks to their open-source model.

Jupyter is one of the most popular open-source notebook platforms. It has millions of users across different disciplines, and there are more than one million public Jupyter notebooks available on GitHub repositories [271]. Jupyter uses language *kernels* to introspect source code and, by default, it comes solely with an *iPython* kernel. Nevertheless, the platform also provides an API for creating language kernels to support new languages. In Jupyter's context, a language kernel provides programming language support, and it is responsible for enabling the communication between the notebook front-end and the desired language. Additionally, it is responsible of executing code, handling computation results and errors, and returning the results to the front-end. A kernel brings together different language-specific components (e.g., type-checkers, interpreters, compilers, formatters, pretty-printers, syntax highlighters).

The development of new language kernels is a cumbersome task; implementing Jupyter's low-level wire protocol requires substantial effort. Language Workbenches (LWBs) offer a default set of generic Integrated Development Environment (IDE) services [99]. Prior research [54, 68, 98, 99, 148, 267, 287] has shown that IDE generation is feasible for Domain-Specific Languages (DSLs). Nevertheless, no support is available for generating language kernels for computational notebooks. The addition of generic language kernels to the LWBs toolbox opens up the notebook metaphor for software languages in a generic fashion.

In this chapter, we present an extended version of a tool demo about Bacatá [356]. Bacatá is a language-parametric notebook generator that implements and hides the complexity of Jupyter's low-level wire protocol. Additionally, Bacatá offers a set of generic hooks for registering language-specific services. We included Bacatá as part of the generic IDE services offered by the Rascal LWB [180]. Thus, creating a language

kernel for a DSL becomes a matter of writing a few lines of code. The implementation of Bacatá, along with documentation and examples is available on Github[1] .

The contributions of this chapter can be summarized as follows:

- We motivate the computational notebook metaphor from the software language engineering perspective. To illustrate this, we provide a Feature-Oriented Domain Analysis (FODA) [164] based on 16 computational notebook platforms (Section 2.2).

- We present Bacatá-Core, a generic language protocol implemented in Java. This protocol simplifies Jupyter's language kernels development process (Section 2.3.2).

- We introduce Bacatá-Rascal, a lightweight bridge between Bacatá-Core and Rascal that enables the communication between Jupyter and Rascal. We highlight how to use this bridge to generate language kernels for DSLs developed using the Rascal LWB (Section 2.4).

- We evaluate Bacatá by generating language kernels for three different languages, implemented in Rascal, namely, Halide* (a subset of Halide [280]) a DSL for digital image processing, SweeterJS an extended version of JavaScript, and QL [99] a DSL for defining questionnaires. Moreover, We measure the amount of reused code for generating a language kernel (Section 2.6).

We conclude the chapter with a discussion of related work and future research directions (Sections 2.7 and 2.8).

2.2 COMPUTATIONAL NOTEBOOKS

Storytelling is a pedagogical strategy and a robust communication and collaboration tool [92]. For instance, computational notebooks enable end-users to teach, learn, and share knowledge. Notebooks are a contemporary style of literate programming [183]. They allow users to interleave documentation, executable source code, and output results in a single linear document.

2.2.1 *Anatomy of a Notebook*

In its purest form, a computational notebook is a cell-based document. There are three different types of cells, namely, documentation, input, and output cells. The first is used to write prose. The second contains executable source code. The last one exposes the result of executing input code.

For instance, Figure 2.1 shows a tiny notebook with three cells. The first row is a documentation cell that contains prose text explaining what is going to happen. The second cell displays an input cell where the user has entered the expression 1 + 2 in

---

1 https://github.com/cwi-swat/bacata

Figure 2.1: Notebook that calculates the addition of two integers.

some programming language. Finally, the last cell shows the output of evaluating the expression.

Furthermore, notebooks are interactive: readers can tweak input parameters, change code snippets, and observe different ways of representing the output. For instance, changing the expression in the input cell will trigger the recomputation of the current output cell. More advanced styles of notebooks feature interactive visualizations of computed results as well, which support interactive exploration of (large) data sets.

Computational notebooks are usually persisted as a single document (e.g., a Jupyter notebook is stored in a JSON-format file), which facilitates sharing. The notebook's results can be relatively easy reproduced since all the documentation, source code, and computed results are part of the same document.

2.2.2 *Notebooks for DSLs*

Most existing language kernels for computational notebooks (e.g., for Python, R, Julia), are based on full-fledged programming languages. DSLs, however, are software languages tailored to solve a particular problem domain. They are designed as a mean of communication between domain experts and software engineers. This fact motivates us to explore if it might be useful to develop notebooks for DSLs. Below we analyze four reasons why DSL users and DSL engineers may benefit from interactive notebooks.

END-USER PROGRAMMING.   Unlike general-purpose programming languages, DSLs are often used by domain experts who are not necessarily proficient in software development or computer science. Interactive notebooks provide a more friendly interface for interacting with source code and documentation than full-fledged IDEs or basic text editors. Additionally, the fact that notebooks run on ordinary web browsers avoids installation hassle. In sum, notebooks make interaction with code less intimidating.

EXPERIMENTATION AND SIMULATION.   Interactive notebooks deviate from the traditional software development setting where the goal is to build production-quality software, towards a setting where exploration and experimentation take center stage. In the context of DSLs, this allows domain experts to experiment with the language,

enjoying immediate feedback and reproducibility. As soon as the design and requirements are stabilized through a DSL, notebooks can provide input to production-level code generators that create the actual software. As such, notebooks reinforce the division of labor between domain engineers and application engineers promoted by Domain-Specific Software Engineering (DSE) [59].

DSL EDUCATION. DSLs are typically small languages, designed for a specific audience. They are developed by smaller teams than general-purpose programming languages like Java or C#. As a result, the use of DSLs incurs costs regarding documentation and training. Computational notebooks can function as live tutorials, providing interactive walk-throughs for a DSL. Notebooks may thus complement standard forms of documentation (e.g., user guides, reference manuals, API documentation), to allow domain experts to familiarize themselves with a new DSL.

LANGUAGE ENGINEERING BENEFITS. The engineering trade-offs in the construction of DSLs are different from general-purpose programming languages. DSLs are often developed in-house, by smaller teams, and require a faster design iteration cycle. Notebooks can provide a valuable tool in the language engineer's toolbox for testing and debugging a language implementation. Especially, since various language engineering aspects can be exposed as part of the notebook. For instance, as we will show in Section 2.6, notebooks can display outputs of language implementation components, such as generated code, static analysis results, and test results.

2.2.3 *Feature-oriented Domain Analysis.*

Computational notebooks are a form of literate programming. There are several platforms for creating them. Each platform offers unique features, yet there are certain features that are shared among them all. We performed a Feature-Oriented Domain Analysis (FODA) [164] to identify common and unique features. Thus, we studied 16 computational notebook platforms (the list of studied tools and the mapping is shown in Appendix A.1), and the result is a feature model shown in Figure 2.2, Figure 2.3, and Figure 2.4. We split the complete feature model into three parts for readability. The first diagram (Figure 2.2) shows an overview of the diagram, while the other two parts display details of two features, namely *Editor* and *Platform*.

The root of the feature model in Figure 2.2 represents the concept of a computational notebook. In the feature model, we only use two kinds of features, namely, *mandatory* for common features (depicted as a box in Figures 2.2 to 2.4), and *optional* for unique features (depicted as a box with a blank circle on top in Figures 2.2 to 2.4). All features are described in Figure 2.2.
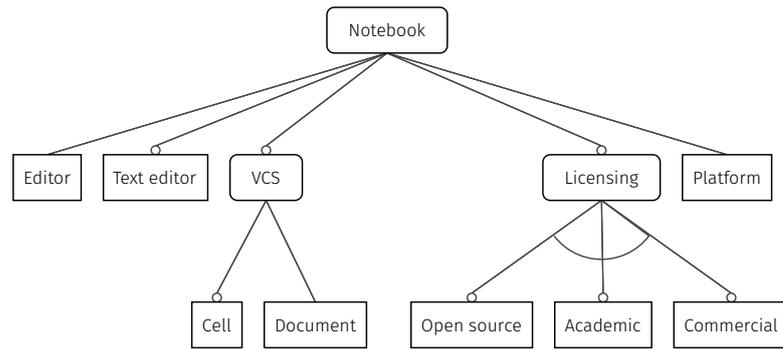
Figure 2.2: Feature model of computational notebooks.
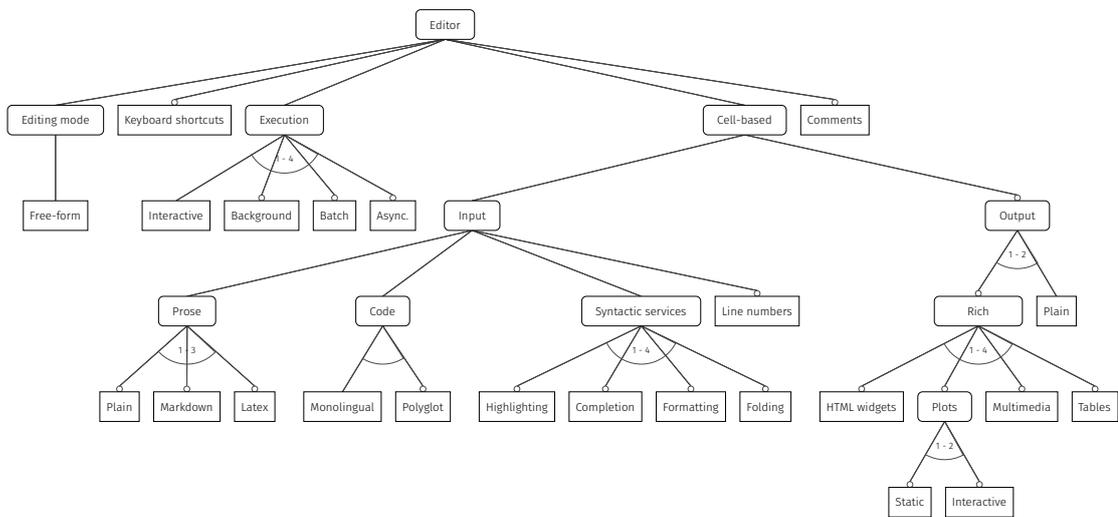
Figure 2.3: Detailed view of the editor feature of the feature model of computational notebooks.

EDITOR.   The notebook editor is the Graphical User Interface (GUI) for creating computational narratives. Figure 2.3 shows a more detailed view of the editor. The Editor feature has five sub-features, namely *Editing mode*, *Keyboard shortcuts*, *Execution*, *Cell-based*, and *Comments*.

EDITING MODE.   The only editing mode supported is *Free-form* in which the users freely edit both code and documentation cells.

KEYBOARD SHORTCUTS.   Keyboard shortcuts is a tool for increasing the productivity of experienced users. Not all platforms consider this a must-have feature, so we consider it as an optional feature.

EXECUTION.   The execution mode is the process of evaluating input code and rendering a response. We found four different modes, and often there is at least one of them in a notebook platform. The default mode is *Interactive*, which means that the response is immediate (not necessarily live [301, 351, 4em]) after pressing the execution button. The other execution modes, included as optional features, are *Background*, *Batch*, and *Asynchronous*. The inclusion of these execution modes is domain-specific. This means that there are domains where these types of processing are needed to satisfy functional and non-functional requirements. For instance, in data science, a big corpus of data is scheduled to be analysed in batch. This is needed given the amount of time and resources required for its processing.

CELL-BASED.   Notebooks are often divided into two types of cells, namely *Input* and *Output*. The first can be either *Prose* in different formats (e.g., plain, markdown, LaTeX) or *Code*. We call a notebook *polyglot* when it allows users to create and execute code cells in different programming languages in a single environment. Traditional IDEs offer a set of *Syntactic editor services* [99] to improve user's productivity and experience. The primary syntactic services offered by notebook platforms are *Syntax highlighting*, *Tab-completion*, *Formatting*, and *Folding*. Finally, *Line numbers* are helpful for error handling and code review. Conversely, output cells are mainly used to display language kernel results (e.g., the result of executing some code cell). These results can be displayed either as *Rich* format elements or *Plain* format elements. Notebooks usually support the following types of rich output media: *HTML widgets*, *Plots* with either static or dynamic content, *Multimedia* (e.g., images, animations), and *Tables*.

COMMENTS.   We found that some platforms allow users to add comments to the notebook itself, and not only as code comments.

TEXT EDITOR.   Text editors are included as part of a notebook platform to edit file documents other than single notebooks. This feature is not present in all the platforms, so it is marked as an optional feature.

VCS.   Version control is fundamental for managing changes, yet there is not a standardised way of tracking changes in a notebook environment. We found two ways of doing versioning of notebooks, either *Document-oriented* or *Cell-oriented*. The former keeps track of all the changes at a document level, there's no notion of cells. The latter keeps track of all the changes at a cell level, which means the VCS show modifications per cell and not per document.

LICENSING.   There are three different types of licenses used by most of the studied notebook platforms, namely, *Open-source*, *Academic*, and *Commercial*.
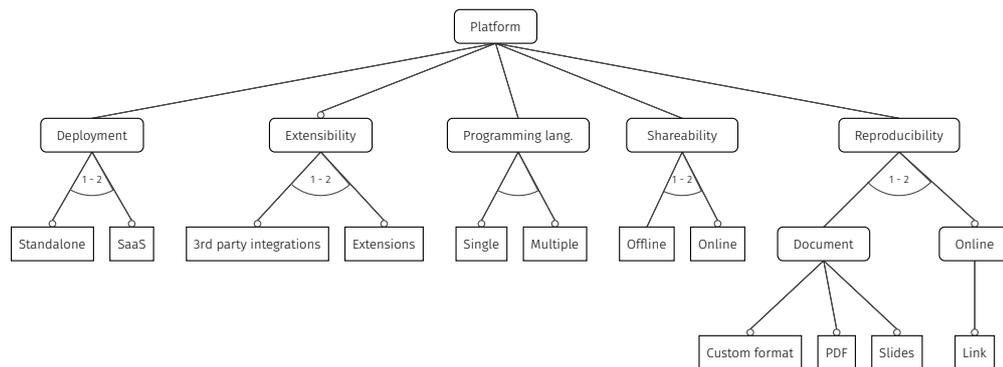
Figure 2.4: Detailed view of the platform feature.

PLATFORM.   It embodies all the additional components of a notebook framework, beyond the editor. We have divided the platform into five sub-features, as shown in Figure 2.4. The meaning of each sub-feature is explained below.

DEPLOYMENT.   Two main deployment models are used for notebook platforms, *Standalone* and *Software as a Service* (SaaS). In the first model, users require a local infrastructure to install and run the platform, so they are responsible for all maintenance activities (e.g., updates, security). In the second model, users do not require a proper infrastructure, and they do not have to install anything. They can use the notebooks straight out of the box, yet the only requirement is a computer with an internet connection and a web browser.

EXTENSIBILITY.   Some platforms come with a lot of plugins and built-in integration, yet some others come with a limited set of features. Platforms of the latter kind allow developers to enhance the default behaviour through a set of APIs. There are two ways of extending these platforms, either by integrating *Third-party* applications or services, or by building *Extensions*.

PROGRAMMING LANGUAGE.   Notebook platforms can support either one or multiple programming languages. In this context, we consider that a platform supports multiple programming languages if it allows users to create notebooks with different programming languages. For instance, MATLAB live editor [142] only supports Matlab as a programming language, while Jupyter supports several programming languages [158].

SHAREABILITY.   Notebooks can be easily shared to have multiple users working on the same notebook. Each user may be focused on different cells in the document. The sharing can be done *online* or *offline*. On the one hand, online sharing means that two or more users can modify the same notebook at the same time, while visualising

modifications made by other users. On the other hand, offline sharing does not allow the modification of the notebook simultaneously. Instead, a user works on his own document, which is then easily shared with other users. This feature encourages collaboration among different people. Sharing capabilities are supported by the single document metaphor adopted by notebooks.

REPRODUCIBILITY.   Notebooks are often used in scientific contexts, so reproducibility is essential for peer review, validation, and verification. Notebooks can contain both the explanation and development of a scientific result and provide the ability to reliably reproduce previous interactive computations using the same data to obtain identical results. There are two different means of sharing a notebook, either by sending a *Document* that contains the notebook or by sharing a *Link* that points out to the notebook. In general, there is no standard format for notebooks, so each platform has its own. However, most platforms allow users to export the notebook's content, including its results (output cells) in different file formats such as PDF or as slides.

Summary

Looking at the feature model, we can observe that some features are language-specific, and some are independent of the actual language. The following features are in the first category: highlighting, completion, formatting, folding, and rich media. The other features are orthogonal to the language-specific features and are handled generically by notebook frameworks such as Jupyter.

Apart from rich media output, perhaps, the language-specific features are already part of the standard toolset of LWBs [99], which opens up the possibility to reuse language components. In the following section, we present Bacatá, and we describe how we generate Jupyter language kernels using a LWB. Besides, we demonstrate how language engineers can reuse existing language-specific components within a computational notebook.

2.3 BACATÁ

Bacatá is a language-parametric interface between the Jupyter platform and the Rascal LWB. It provides a mechanism to generate Jupyter language kernels in a way that language engineers can reuse existing language components such as grammars, parsers, type checkers, code generators, and interpreters. According to Jupyter's documentation [159], a language kernel *"is a program that runs and introspects the user's code"*. In other words, it is a language component that is capable of both evaluating an input piece of code and resolving object types at runtime.

In the reminder of this section, we detail Bacatá's design and implementation. First, we present its architecture; second, we explain in detail Bacatá's language service

interface (Bacatá-Core) and Bacatá-Rascal; and third, we show an example of how to generate a language kernel for a toy language using Bacatá.

2.3.1   *Architecture*
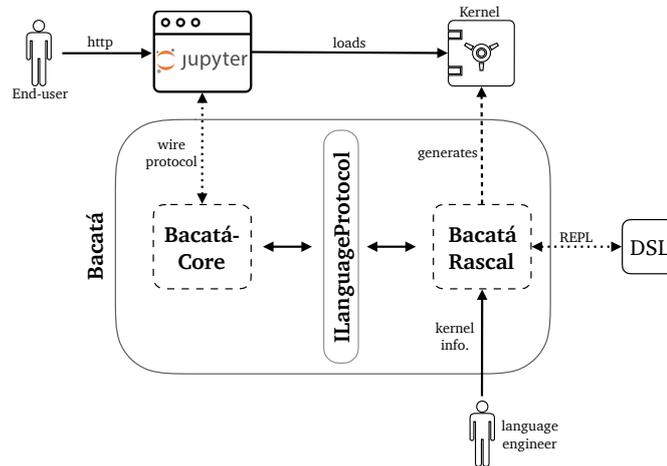
Figure 2.5: Bacatá's general overview architecture.

Figure 2.5 shows Bacatá's architecture general overview. This diagram highlights its most essential components and users (roles). There are two different Bacatá roles, namely *language engineer* and *end-user*. Language engineers use Bacatá to generate language kernels, whereas users interact with the generated kernel through the Jupyter notebook interface.

Bacatá is divided into two components, Bacatá-Core and Bacatá-Rascal. Bacatá-Core is responsible for enabling the communication between the Jupyter platform and a LWB. It exposes the `ILanguageProtocol`, which is a generic language protocol interface (comparable to Microsoft's Language Server Protocol (LSP) [200]). This interface can be implemented to work with any LWB because it is independent of Rascal. Bacatá-Core is also responsible for collecting the user's input code and sending it to the respective language interpreter.

On the contrary, Bacatá-Rascal is Rascal-dependant. It implements the `ILanguageProtocol` interface and provides the means for connecting Rascal-based languages to Bacatá-Core. A `Kernel` type is Bacatá-Rascal's entry point. It is an Algebraic Data Type (ADT) designed to capture some required language-specific information. This ADT is actively used to complete two essential tasks in the kernel generation. First, it generates the language kernel; and second, it is the input for generating language-specific artefacts, such as syntax highlighter modes and tab-completion functions.

Bacatá's language kernels are registered automatically as part of Jupyter's supported languages (in the current environment). Therefore, these language kernels also become available to be used through any of the Jupyter's interfaces.

End-users do not interact directly with Bacatá; from their perspective, Bacatá-Core and Bacatá-Rascal are hidden components. They have to choose a language kernel from Jupyter's notebook interface, and afterwards they can start interacting with the language. When a language kernel is chosen, Jupyter makes a callback to Bacatá to instantiate a language Read–Eval–Print Loop (REPL). A language REPL is a language artefact capable of reading expressions, evaluating them, and printing the result of their evaluation. This REPL is needed to execute user's code in the desired language.

2.3.2 *Bacatá-Core*

Jupyter's mechanism to support new languages is through language kernels. A language kernel is a program that executes the user's code and implements the *wire protocol* [157]. This protocol can be implemented in two ways, by creating a Python wrapper kernel or by hand. The former reuses all the IPython [268] kernel infrastructure and is meant to be used by languages with Python bindings (e.g., Hy ²) [160]. The latter has to be implemented by hand in the target language, yet it gives language engineers full control. In this chapter, we focus on the second approach since it gives full control from the language engineering point of view.

The *wire protocol* is Jupyter's communication protocol, and it is implemented using ZeroMQ sockets [4]. Bacatá-Core is responsible for implementing it. This protocol involves five different types of sockets (control, heartbeat, IOPub, shell, and stdin) and more than a dozen messages.Each message with its structure can be sent only through a specific socket. Each socket and message has just one responsibility, but the right composition allows the interaction between Jupyter's notebook interface and language kernels. Given the number of sockets and messages, its implementation is error-prone and difficult to debug. The protocol is implemented at the session and presentation layer in the OSI model [396].

To prevent language engineers from having to deal with Jupyter's protocol, Bacatá-Core offers an abstract class called `MetaJupyterServer` and an interface called `ILanguageProtocol` (Appendix A.3). `MetaJupyterServer` hides all the complexity of the wire protocol, including its socket management; while the `ILanguageProtocol` focuses on the language engineering aspects, such as user's code evaluation, code fragments auto-completion, and statements completion.

Bacatá-Core is LWB-agnostic. To use Bacatá with a new LWB, language engineers have to fulfill two conditions. First, they need to implement `ILanguageProtocol`. Second, they have to implement the abstract methods of the `MetaJupyterServer` class (further details on the abstract methods are shown in Appendix A.2).

---

2 https://docs.hylang.org/en/stable/

In the next section, we discuss implementation details and design decisions of the `ILanguageProtocol` and `MetaJupyterServer`. We also show the integration with the Rascal LWB and how to generate language kernels in a language-parametric way.

2.4 BACATÁ-RASCAL

Language kernels are Jupyter's mechanism to support new languages. Bacatá-Rascal is a language kernel generator for Rascal DSLs. To enable the communication between a Rascal DSL and Jupyter, we implement a Java class `DSLNotebook` that extends the abstract class `MetaJupyterServer`. `DSLNotebook` provides a definition of the language-specific methods declared in the `MetaJupyterServer` class, such as code completion and code execution. Additionally, it requires the definition of an *interpreter*. This interpreter is a generic implementation of the `ILanguageProtocol`, which is used to introspect and evaluate user's code for all Rascal DSLs. It is designed as a language-parametric mechanism for hooking the DSL's REPL into Bacatá. Concretely, the generic interpreter takes as input the DSL's source code path and the REPL's qualified name. Bacatá-Rascal uses this information to load the language and to execute the user's code in the desired language.

For creating a new language kernel with Bacatá, language engineers have to define an interpreter for their language. To implement an interpreter they have to instantiate a REPL ADT (Listing 2.1). This REPL object is the language's interactive interpreter. It is used to evaluate the user's input code (using the `handler` function) and to complete code fragments (using the `completor` function). The result type of each function (`completor` and `handler`) is also shown in Listing 2.1, lines 4-8.

Listing 2.1: REPL type definition.

```
1  data REPL
2    = repl(Result(str) handler, Completion(str) completor);
3
4  alias Completion
5    = tuple[int position, list[str] suggestions];
6
7  data Result
8    = text(str result, list[Message] messages);
```

The walk-through to create a language kernel for a DSL using Bacatá is explained below.

1. Language engineers have to specify language's information through a value of type `Kernel` (Listing 2.2). This ADT defines configuration parameters for obtaining language-specific information (e.g., name and location of the logo of the language) required either by Bacatá-Core or Bacatá-Rascal. Additionally, it contains informa-

tion about relevant resources such as the language's project path and the REPL's qualified name.

2. The language engineer calls `bacata` (Listing 2.4), which has two overloaded function definitions. On the one hand, the first definition (Listing 2.4, line 1) takes an argument value of type `Kernel` (Listing 2.2) and two optional boolean parameters, namely, debug and Docker [234]. The first, as its name suggests, is used for language kernel debugging. The second is used to generate a *Dockerfile* that assembles all the required dependencies to run the generated computational notebook (including the generated language kernel). The second `bacata` definition (Listing 2.4, line 2) has an extra parameter, namely `grammar`. We included this parameter in case the language engineer wants Bacatá to offer syntax highlighting support through CodeMirror modes. We explain syntax highlighting details in Section 2.5.1.

3. As a result of calling `bacata`, there are several side-effects. First, Bacatá verifies Jupyter's correct installation and the definition of the required environment variables. Second, it generates a JSON serialised dictionary (a.k.a. *kernel.json*) that contains language-specific information (including Bacatá's wiring) and Jupyter's connection details (e.g., ZMQ socket ports). Third, it constructs a value of type `Notebook` (Listing 2.3) that encapsulates either two or three functions (depending on the selected overloaded constructor). The `serve` function starts Jupyter's server, while the `stop` function is used to shut down the server. Moreover, to capture Jupyter's server logs, we use the `logs` function. Fourth, it installs the generated language kernel in the current Jupyter environment. Finally, to obtain an updated version of the front-end, Bacatá automatically recompiles all Jupyter's assets (including DSL-specific artefacts such as generated CodeMirror modes).

Listing 2.2: Kernel type definition.

```
data Kernel
  = kernel(str languageName, loc projectPath, str replFunction, loc logoPath = |tmp:///|);
```

Listing 2.3: Notebook type definition.

```
data Notebook
      = notebook(void() serve, void() stop)
      | notebook(void() serve, void() stop, void() logs);
```

Listing 2.4: Bacata function.

```
1 Notebook bacata(Kernel kernel, bool debug = false, bool docker = false) {…}
2 Notebook bacata(Kernel kernel, type[&T <:Tree] grammar, bool debug = false, bool docker =
      false) {…}
3 Notebook bacata(Kernel kernel, Mode mode, bool debug = false, bool docker = false) {…}
```

2.5   CONCRETE EXAMPLE: THE CALC LANGUAGE

So far, we have introduced Bacatá's components; now, we are going to explain how to generate a language kernel using Bacatá for a simple calculator language (CALC). This language already existed and it was already implemented using the Rascal LWB. However, all the Rascal code could also be written in Java or using other LWB. We present Calc's grammar in Listing 2.5. It consists of commands (Listing 2.5, lines 5-7) and expressions (Listing 2.5, lines 11-15). There are two types of supported commands, namely assignments and expression evaluation. Calc's expressions are variables, numbers, multiplication, and addition. To execute commands, there is an exec function (Listing 2.5, line 19) that returns a tuple containing an integer, and a possibly updated environment (value of type Env, Listing 2.5, line 18). Finally, expressions evaluate to numbers (eval function in Listing 2.5, line 21).

Listing 2.5: CALC's grammar definition using Rascal's built-in formalism.

```
1  module Syntax
2  extend lang::std::Id;
3  extend lang::std::Layout;
4
5  syntax Cmd
6    = Id "=" Exp
7    | Exp;
8
9  lexical Num = [\-]?[0-9]+;
10
11  syntax Exp
12    = Id
13    | Num
14    | left Exp "*" Exp
15    > left Exp "+" Exp;
16
17  alias Env = map[str, int];
18
19  tuple[int, Env] exec(Cmd cmd, Env env) { … }
20
21  int eval(Exp exp, Env env) { … }
```

Based on this existing language definition, we now explain how to get a REPL (Listing 2.6). calcREPL returns a value of type REPL (Listing 2.1). As explained before, the REPL requires the definition of two functions, *handler* and *completor*. First, Calc's handler is shown in Listing 2.6 (lines 7-15). It takes the user's input as a parameter and tries to parse it. If the parsing phase is successful, it proceeds to execute the parsed command and returns a text(Listing 2.1, line 8) result (line 11, Listing 2.6). Otherwise, if there is a parsing error, the function (calcHandler) returns an error message with an empty result (line 14, Listing 2.6). Second, calcCompletor implements a straightforward

completion function. It iterates over all the variables stored in the current environment (env, line 5 Listing 2.6), and returns the set of variables that match with the prefix parameter. Finally, line 22 (Listing 2.6), we construct and return a value of type REPL containing the functions mentioned above (calcHandler and calcCompletor).

Note that code in Listing 2.5 and Listing 2.6 is entirely independent of Bacatá. Therefore, all the code can be reused outside the notebook environment. For instance, the concrete syntax definition, the evaluation, the execution, and the REPL functions may all be used in a standalone IDE for Calc.

In particular, in Listing 2.7 we detail how to generate a Jupyter language kernel with Bacatá. We use the REPL function calcRepl defined in Listing 2.6. We first create a Kernel value, consisting of the language's name, the project's path, and the REPL's function qualified name. Bacata's function (line 4 in Listing 2.7) returns a Notebook value, and as a side effect, it generates a *kernel.json* file. The Notebook value may be used to start Jupyter's notebook server within the same session. Alternatively, it can also be started from the command-line outside Rascal Eclipse.

Listing 2.6: A REPL implementation for CALC.

```
1  module Repl
2  import Syntax;
3
4  REPL calcREPL() {
5    Env env = ();
6
7    Result calcHandler(str line) {
8      try {
9        Cmd cmd = parse(#Cmd, line);
10       <n, env> = exec(cmd, env);
11       return text("<n>", []);
12     }
13     catch ParseError(loc l):
14       return text("", [message("Parse error"
           , l)]);
15   }
16
17   Completion calcCompletor(str prefix)
18     = <pos, [ x | x ← env, startsWith(p,
           x) ]>
19     when /<p:[a-zA-Z]*$/ := prefix,
20       pos := size(prefix) - size(p);
21
22   return repl(calcHandler, calcCompletor
         );
23 }
```

Listing 2.7: Rascal's interactive session.

```
1  > k = kernel("Calc", |project://Calc|,
2      "Repl::calcRepl");
3  >> …
4  > nb = bacata(k);
5  >> …
6  > nb.serve();
7  The notebook is running at:
8    |http://localhost:8888|
```

Listing 2.8: Syntax Mode data type.

```
data Mode
  = mode(str name, list[State] states);


data State
  = state(str name, list[Rule] rules);


data Rule
  = rule(str regex, list[str] tokens,
       str next = "", bool indent = false,

       dedent = false);
```

2.5.1 *Input Cells: Syntax Highlighting*

Jupyter's input cell editor is based on the CodeMirror editor [131], thereby for syntax coloring, Jupyter uses CodeMirror *modes*. They are similar to so-called "Textmate grammars"[3] that are used by editors such as Textmate, VS Code, SublimeText, and many others. Listing 2.8 shows the definition of the `Mode` type. A `Mode` is defined by a name and a list of states, and each state has a name and several rules that apply to each state. Finally, a `Rule` defines a regular expression that matches a particular substring and assigns a list of tokens that determine its visual appearance. After a rule has matched, it may transit to another state via the `next` property. There are two optional Booleans `indent` and `dedent`. They are responsible for controlling auto-indentation in block constructs.

Bacatá allows language engineers to describe and generate CodeMirror modes in an automatic or manual fashion. The first approach is a built-in feature of Bacatá. It takes the language's grammar and analyzes it to generate simple modes for keyword highlighting using reflection. The second approach is by manually defining a `Mode`, and sending it as a parameter to the `bacata` function. A simple mode for the CALC language is implemented as follows:

```
Mode calcMode =
 mode("Calc", [state("ini", [rule("[0−9]+", ["number"]), rule("[a−zA−Z][a−zA−
     Z0−9_]*", ["variable"])])]);
```

This mode defines a state with two rules, one for numbers and the other one for variables. To create a CALC notebook using the `calcMode` mode, one can call the `bacata` function as shown bellow:

```
bacata(k, calcMode);
```

2.5.2 *Output cells: Interactive Visualizations*

Jupyter's notebook interface runs in the browser, so this allows cells to contain arbitrary HTML/CSS/JS widgets, beyond the plain output shown in Listing 2.6 (line 11) for the CALC language. Bacatá supports fully interactive, stateful graphical user interfaces as output cells. Said support is achieved through the integration between Bacatá and Salix [326], Rascal's web UI framework. Salix offers support for standard HTML and SVG elements, and integration with several graph frameworks and chart frameworks such as DagreJS[81] and Google Chart [123].

Salix emulates *Elm*'s architecture[4]; it is divided into three parts, namely *model*, *update*, and *view*. In Salix, these three pieces are encapsulated as an `App[&T]` type, where `&T` represents the application data *model*. An `App` encloses a *view* to draw UIs, and the

---

3 https://manual.macromates.com/en/language_grammars
4 https://guide.elm-lang.org/architecture/

update function to update the application's model when the user triggers an event. The way Bacatá uses Salix Apps is by allowing its usage as a REPL output. To achieve this, we extend the definition of the Result type (Listing 2.1) as follows:

```
data Result
  = ...
  | app(App[&T] app, list[Message] messages);
```

As a result of extending the Result type definition, a REPL can return fully functional stateful output cells, leveraging all UI features of Salix.

Now we are going to extend our CALC language with a tiny expression debugger. This debugger is used as a way of interactively debugging variables values, and see the effect of changing them in the evaluation of expressions. The following code snippet reflects the required changes to integrate the expression debugger in the current implementation of the CALC language. First, we add a new production rule to the Cmd non-terminal that triggers the debugging visualization as follows:

```
syntax Cmd
  = ...
  | "show" Exp;
```

As a result of the previous change, when the user types and executes the expression **show** x + y, the resulting output cell will contain a debugger of the expression x + y.

Listing 2.9 shows the implementation of the expression debugger for the CALC language using Salix. The application model for this debugger is the environment Env and the Msg type, binds the unique event, which keeps track of changes in the variable's value. In Listing 2.9 (Lines 3 to 20) we show a function named expDebugger that takes as arguments an expression Exp and an environment Env and produces a Salix application (App[&T]). This type encapsulates three functions, namely, init, view, and update. The first one initializes the application model. The second function takes the current environment and draws the UI based on that information. The UI shows a textual representation of the expression, including its computed value. Also, for each variable in the environment, the UI creates a label and a slider. Finally, the last function is responsible for updating the model.

Finally, to complete our expression debugger, we have to include it as part of the CALC's REPL. As said before, we want to display the debugger whenever a user executes a **show** command. To achieve this, we added the following **if**-statement just after parsing the user's input code (Listing 2.6, line 9).

```
Cmd cmd = ...
if ((Cmd)'show <Exp e>' := cmd) {
  return app(expApp(e, env), []);
}
```

The last **if**-statement uses Rascal's concrete syntax pattern matching to check whether cmd is a **show** command or not. If cmd is a **show** command, it binds e to the argument

expression. If the match succeeds, it returns a value of type `Result` (using the `app` constructor) that encapsulates the Salix application defined in Listing 2.9.

Listing 2.9: Expression debugger defined using Salix.

```
data Msg = var(str x, str val);

App[Env] expApp(Exp e, Env env) {
  Env init() = env;
  void view(Env env) {
    div(() {
      for (str x ←env) {
        text("<x>: <env[x]>");
        input(type("range"), value(env[x]),
              onInput(partial(var, x)));
      }
      text("<e>: <eval(e, env)>");
    });
  }
  Env update(var(x, v), Env env)
      = env + (x:toInt(v));
  return makeApp(init, view, update);
}
```
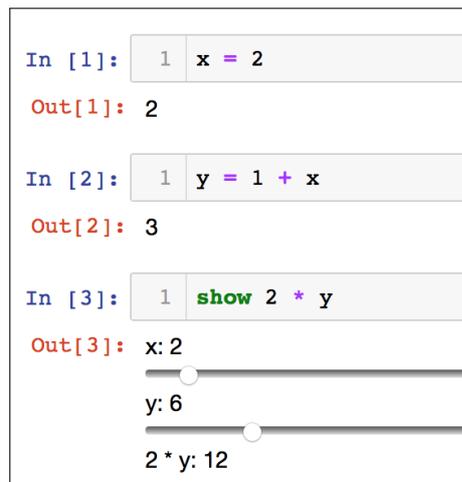

Figure 2.6: CALC notebook that includes an interactive debugger for expressions.

Figure 2.6 displays a CALC notebook, including a debugging interface (output cell 3). In this notebook, a user has defined two variables, x and y. Then a `show`command was executed to debug the effect of changing the value of the current variable bindings on the expression 2 * y. When the user changes the slider for y, the new result is simultaneously updated.

2.6 CASE STUDIES

We have used Bacatá to generate notebook interfaces for three different languages, namely, Halide*, SweeterJS, and QL. All these languages have been implemented using the Rascal LWB and are available on GitHub [358].

2.6.1 *Halide**

Halide [280] is an embedded language for image processing and computational photography. Bacatá requires a grammar and a REPL written in Rascal to create a notebook. However, for Halide, we did not have an existing grammar nor REPL. Therefore, we implement Halide*, a Halide grammar, and a REPL that enables the interaction with the DSL through a notebook interface. Halide* captures a subset of the Halide language. It is important to remark that the notebook way of working influences the design of the Halide* 's grammar and the REPL. Also, the current implementation was written in Rascal, but it could have been done in Java or using other LWB. Halide* was designed for Océ, a Canon company that develops, and manufactures printing and copying hardware. As part of their development process, one of their needs is the construction of digital image processing algorithms. In this process, there are people with different backgrounds (e.g., mathematicians, physicists, electrical engineers) not necessarily with a background in computer science. However, most of them were already familiar with a notebook way of working. They wanted a mechanism to implement said algorithms in a notebook environment that speeds up their development cycle.

Halide* divides the program into three different categories, namely, data loading, algorithms, and execution. The data loading category includes the language constructs used to load data into buffers (e.g., images, arrays); the algorithmic category describes the data transformation the user wants to express algorithmically (e.g., blur, gradient); finally, the execution category takes the data and applies algorithms over it.

Halide* generates, compiles, and executes native Halide source code; the Halide compiler is responsible for the compilation and execution steps. We introduced some syntactic sugar to the Halide grammar to detect particular language constructs using the categories mentioned above. Mainly, we added function wrappers to differentiate between main functions, image pipeline definitions, compilation strategies (e.g., ahead of time or just in time compilation), and execution. Halide*'s cell execution is performed through the REPL in two steps. First, we compile Halide* code into Halide code, then Bacatá delegates the compilation and execution process to g++. Bacatá intercepts those results, parses them into HTML, and then displays them within the output cells of the Halide* notebook.

A prototypical session using the Halide* notebook is shown in Figure 2.7. It highlights the visualization of multimedia results and inspection of artifacts generated by the compiler. In Figure 2.7a, the user loads a png image (as shown in the output cell [1]).

(a) Loading an image and defining a function `blur` (snipped).

(b) Calling `blur` and inspecting generated artifacts.
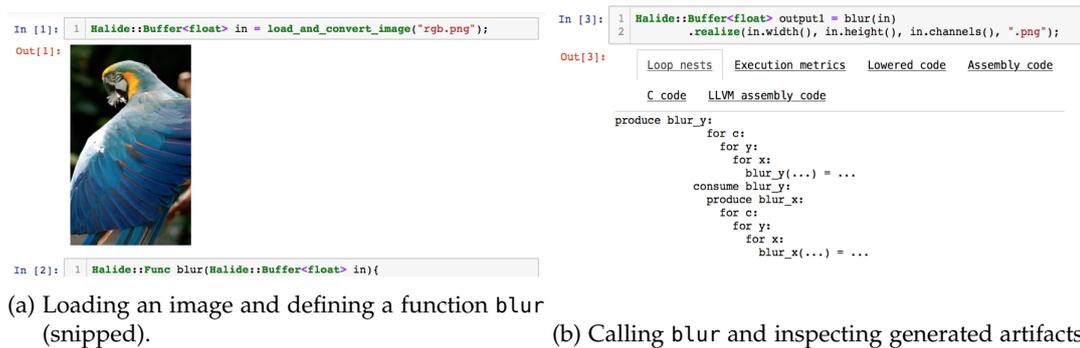
Figure 2.7: Halide notebook.

Then a `blur` function is defined in the input cell [2], which does not produce an output but is now available for use. Then, in Figure 2.7b, the `blur` function is invoked on the input image `in`. The result shows a tabbed interface built to inspect loop nesting, execution metrics, lowered code, assembly code, C code, and LLVM assembly code. Alternatively, the resulting image can be shown.

2.6.2 *SweeterJS*

SweeterJS [80] is a framework for language extension of Javascript (ECMAScript 5), and it is used to teach source-to-source transformations (desugaring) using Rascal. SweeterJS was already implemented as a Rascal language, although it did not have a REPL. Therefore, we created a SweeterJS's REPL that mostly reuses the existing SweeterJS IDE. To illustrate the benefits of notebooks from the language engineering perspective, we have generated a notebook interface for SweeterJS. The notebook interface allows students to experiment with the language by writing, executing, and transforming SweeterJS's code snippets. Therefore, students obtain the computed result and the desugared version (ECMAScript 5) code.

Figure 2.8 shows a SweeterJS notebook that contains as input some JavaScript code with an SQL-like query expression (`In[16]`, line 5). Furthermore, it also shows the result of executing the desugared version of the code and the desugared code itself. For instance, the query expression (line 5 in the input cell) is transformed into a JSLINQ query constructor (`Out[16]`, lines 5-7).

2.6.3 *Questionnaire Language (QL)*

QL is a DSL for defining interactive questionnaires that has been used to benchmark and evaluate LWBs [99]. Indeed, QL is interesting from the notebook metaphor perspective since QL specifications define interactive GUI forms. There is already a Rascal imple-

```
In [16]:    1  // Select query demo
            2  var myList = [{Name:"Chris",Surname:"Bell"},
            3               {Name:"Joe",Surname:"Ross"},];
            4
            5  var q = select Name from myList where Name === "Chris";
            6
            7  console.log("Query output: ");
            8  console.log(q);

Out[16]:
            Desugared JS source     Console output

            1  // Select query demo
            2  var myList = [{Name:"Chris",Surname:"Bell"},
            3               {Name:"Joe",Surname:"Ross"},];
            4
            5  var q = JSLINQ(myList)
            6    .Where(function(item) { return item.Name === "Chris"; })
            7    .Select(function (item) { return {Name: item.Name}; });
            8
            9  console.log("Query output: ");
           10  console.log(q);
```

Figure 2.8: Desugared output from a SweeterJS notebook.

mentation of QL [325], so like we did for SweeterJS, we reused the existing language definition and IDE for QL to built on top of that a REPL. In particular, a questionnaire consists of a *form* that may contain one or more *questions*, and each question has a *type*. There are three different types of questions, namely labeled, conditional, and computed. QL programs in a notebook environment can be visually represented as interactive HTML widget forms implemented using the Salix library. Besides, the QL notebook supports the visualization of control dependencies between questions, which is a valuable feature for questionnaire designers to understand the conditional logic of a questionnaire.

Figure 2.9 shows a QL notebook for a simple tax filing questionnaire. The user first defines a questionnaire myForm using the form-command (Figure 2.9a). Then, in Figure 2.9b, the form is rendered as an interactive HTML widget using the html command. Note that the output is a fully working questionnaire, as if we have deployed it in a production environment. Thus, this supports testing interactive questionnaires at design time. Alternatively, to understand the conditional logic of a form, the user can visualize the control dependencies using the visualize command (Figure 2.9c).

2.6.4 *Effort*

To assess Bacatá's flexibility for creating Jupyter language kernels, we compare the number of Source Lines of Code (SLOC) that are Bacatá independent versus the number of SLOC required to define the notebook itself. These results are shown in Table 2.1.

The CALC language is included as a baseline, and it is the code discussed in Section 2.5. On the one hand, the reused code that is independent of Bacatá is the grammar

(a) QL form definition.

(b) QL form execution.

(c) Question control dependencies.
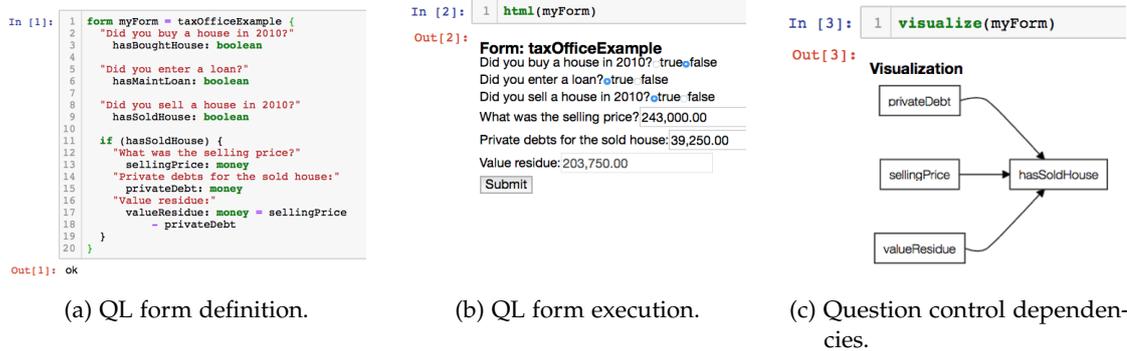
Figure 2.9: Tax filing questionnaire using a QL notebook.

Table 2.1: Number of reused SLOCS and notebook-specific SLOCS.

Language	Reused SLOC	Notebook SLOC
Calc	37	50
Halide*	51	647
SweeterJS	579	162
QL	771	120

definition and the `exec` and `eval` functions. On the other hand, the notebook specific code includes the definition of the REPL and the expression debugger.

The Halide* implementation differs from the other case studies because we implemented the whole language from scratch. We reverse-engineered the Halide language to design Halide* and make it notebook-friendly. The amount of notebook specific code is higher than the reused code because it required a reengineering process of an existing non-notebook-friendly language. It required changes in several language components. The syntax definition is the only completely reusable component. When implementing the Halide* notebook, we have found some language-specific design considerations that should be addressed to change an existing language into a notebook-friendly language, but these details are out of the scope of this chapter.

In the remaining case studies, namely SweeterJS and QL, the ratio between reused and notebook specific code is much higher. In the case of SweeterJS, the reusable code includes JavaScript's syntax definition, language extensions for state machines, queries, and a variant of HAML [337], and the required transformations to desugar language extensions to vanilla JavaScript.

In addition, the REPL for QL was already defined and it included a Salix visualization, so that we could reuse it automatically. The same holds for other language components such as syntax definition, name resolution, and type checking. The new code includes

the code for the REPL and the control-dependency visualization. However, this code can also be used outside the notebook environment.

Table 2.1 shows that creating a language kernel for Jupyter notebook using Bacatá requires limited effort. Its main requirement is a REPL definition, which in some cases is as simple as wiring some existing components together as we did with QL and SweeterJS. In contrast, other languages may require more profound language modifications due to the different execution models. However, even in those cases, language engineers benefit from the Jupyter's protocol implementation.

Bacatá can be used across several domains by various DSLs. In this chapter, we used it for four languages, namely Calc, Halide*, SweeterJS, and QL. Therefore, we can conclude that Bacatá is functional and effectively applicable to various domains and languages. However, as discussed above, the benefit of using Bacatá may differ between domains and languages. For instance, the Halide* notebook required more effort due to the language reverse-engineering and REPL development. While, in Calc, SweeterJS, and QL, we reused the existing DSL machinery.

2.7 RELATED WORK

In the context of software language engineering, tooling support is vital to increase and improve the adoption of language-oriented programming. Bacatá contributes to the research line on program environment generation [54, 68, 98, 99, 148, 267, 287].

Fowler [110] popularized the term LWB. In one of his essays, he described the foundations of language-oriented programming, its benefits, and drawbacks. Also, he explains the critical role of IDE tool support in language-oriented programming. Primarily, proper tool support for DSLs may reduce the learning curve and boost their viability.

LWBs make use of metalanguages and meta-programming techniques that reduces the costs of building DSLs and its tooling. Our work aims to follow the same trend by offering a mechanism of generating a computational notebook interface based on a language specification. Mainly, notebooks offer a different GUI for interacting with code and documentation. This interface does not clash with the more traditional GUIs such as IDEs or text editors, but it offers an enhanced interface mainly for end-user programming [184] and exploratory programming [39, 172, 303].

Interactive computing has emerged as another software development paradigm. So far, researchers have highlighted its importance and benefits for programming related tasks [72, 253]. Cook [72] highlights the main benefits of using interactive programming in software development. He illustrates the main differences between interactive and non-interactive programming. Likewise, Nagar [253] presents a case study about using Python in an interactive computing setting. Mainly, he highlights the importance and value of being able to experiment with code; mostly, the capability of executing

commands and expressions, and its output. Also, the impact it has on the programming language learning curve for end-users.

Computational notebooks integrate different research areas such as literate programming, interactive computing, and end-user programming. However, the primary rationale probably is literate programming [183, 281, 308]. The basis of a computational notebook is the capacity of writing executable source code and narrative text. Also, the ability to document and explain results using multimedia formats such as charts and visualizations.

Moreover, one of the benefits of using notebooks is its sharing capabilities [317]. Turner et al. [349] explored them as an appropriate mechanism for supporting cooperative work and sharing information with non-technical staff. Bacatá follows this direction because we are getting DSLs closer to end-user programmers through a notebook interface.

Currently, there are several studies tackling usability, cognitive, and reproducibility aspects of computational notebooks. For instance, in reproducibility [189, 270, 271, 309], education [258], exploratory programming [39, 133, 172, 173], documentation [388], and data journalism [393]. However, no attention has been paid from the language engineering point of view. There is some work about interactive DSL usage, e.g., domain-specific debuggers [52] and live DSLs [301, 344, 351].

2.8 CONCLUSIONS & FUTURE WORK

Computational notebooks offer a different GUI for interacting with prose, executable source code, and interactive feedback. Contrary to traditional IDE and text editors, notebooks focus on a different way of working focused on computational storytelling and end-user programming (e.g., exploratory programming and data science).

To better understand computational notebooks and their features, we conducted a FODA in which we studied 16 notebook platforms. Based on our findings, we created a feature model that depicts both common and unique features of these platforms.

Jupyter is one of the most popular open-source notebook platforms. Therefore, in this chapter we implemented our approach in the Jupyter context. In general, developing a new language kernel for Jupyter requires much effort. In the context of DSL's, it is even more expensive because their design and implementation cycle is different from general-purpose programming languages. Thus, we introduce Bacatá, a language-parametric kernel generator for Jupyter notebooks. These language kernels reuse existing language components, such as language processors, language formalisms, and type checkers. Thus, implementing a notebook interface for a new language becomes a matter of writing a few lines of code that wire language components together as a REPL.

Moreover, we present Bacatá's architecture and how we implement it within the Rascal LWB. In addition to the default features offered by Jupyter (code execution,

code tab-completion, and syntax highlighting), we integrate Rascal's web-based GUI framework (Salix) to support fully interactive output cells.

We use Bacatá-Rascal to define language kernels for three languages, namely Halide*, SweeterJS, and QL. With these case studies, we exercised multiple aspects of the framework with different kinds of DSLs. As a result, we find that embedding a language into a notebook setting may have an impact on its design. In Section 2.6.4, we compare the number of reused SLOCs versus the number of notebook-specific SLOCs and we observed that Bacatá-generated language kernels require little effort. All the effort is focused on language-specific tasks and not in notebook-specific tasks.

As an on-going part of this project, we plan to study in more detail what are both the limitations and consequences of embedding a DSL into a notebook ecosystem. Moreover, we want to evaluate if it is feasible to embed visual languages into this setting. Finally, we plan to consolidate Bacatá's interface (`ILanguageProtocol`) with Microsoft's LSP [200]. This consolidation would allow language engineers to implement a single interface once and for all.

A PRINCIPLED APPROACH TO REPL INTERPRETERS

Read-eval-print-loops (REPLS) allow programmers to test out snippets of code, explore APIs, or even incrementally construct code, and get immediate feedback on their actions. However, even though many languages provide a REPL, the relation between the language as is and what is accepted at the REPL prompt is not always well-defined. Furthermore, implementing a REPL for new languages, such as DSLs, may incur significant language engineering cost. In this chapter we survey the domain of REPLs and investigate the (formal) principles underlying REPLs. We identify and define the class of sequential languages, which admit a sound REPL implementation based on a definitional interpreter[1], and present design guidelines for extending existing language implementations to support REPL-style interfaces (including computational notebooks). The obtained REPLs can then be generically turned into an exploring interpreter, to allow exploration of the user's interaction. The approach is illustrated using three case studies, based on MiniJava, QL (a DSL for questionnaires), and eFLINT (a DSL for normative rules). We expect sequential languages, and the consequent design principles, to be stepping stones towards a better understanding of the essence of REPLs.

3.1 INTRODUCTION

"The top level is hopeless", Matthew Flatt[2]

Read–Eval–Print Loops (REPLs, also known as command-line interfaces, or interactive shells) are a popular way for programmers to interact with programming languages. They allow incremental definition of abstractions, testing out snippets of code with immediate feedback, debugging executions, and explore APIs.

Some languages, such as scripting languages or interpreted languages, are more naturally compatible with the REPL mode of interaction and the styles of programming that it enables (and that programmers have come to expect). For example, a sequence of valid code snippets written in the REPL of Python can be itself a valid Python program. On the other hand, JShell, for instance, allows programmers to write expressions, statements, variable declarations and method declarations as code snippets, even though these constructs are not allowed at the top-level in Java programs.

Consider the following example JShell interaction (every line is a code snippet sent separately):

---

1 A definitional interpreter is an interpreter that simultaneously defines and implements the operational semantics of a language.

2 https://gist.github.com/samth/3083053

```
class Example {}
Example obj = new Example();
class Example { public int meth() { return var; } }
int var = 1;
```

This example raises the questions whether classes can be redefined, whether `obj` can be accessed after `Example` is redefined or if `obj` is migrated, and, if so, what methods it has and, if `meth` is available, whether a call `obj.meth()` returns `1`. Without giving answers here, the example shows that the relation between a programming language and the behavior of its REPL is not immediately obvious. Matthew Flatt's *ceterum censeo* quoted above bears witness to the fact that the relation can actually be strenuous and cause a lot of confusion. The above questions are fundamentally about language design: several sensible answers are possible and the answers have a significant impact on programmer experience.

In some sense, JShell can be seen to implement *its own* language, which, even though strongly reminiscent of Java, is markedly different. In this chapter, we take this observation and run with it: we assume that a REPL interpreter for $\mathcal{L}$ effectively defines its own language $\mathcal{R}$, often as an extension or modification of $\mathcal{L}$, whose programs are sequences of valid code snippets according to the REPL.

To this end we identify and define the class of languages that underlie REPL interpreters as *sequential languages*. The essence of sequential languages is that the *concatenation of two programs is again a program*. Or, to put it more precisely, a language is sequential if it features an associative sequencing operator $\,\mathbf{\S}\,$, such that the following equation holds:

$$\llbracket p_1 \mathbin{\S} p_2 \rrbracket = \llbracket p_2 \rrbracket \circ \llbracket p_1 \rrbracket$$

The meaning of a sequence of program fragments is defined by composing the meanings of the individual fragments, including any impure effects of these fragments.

The notion of sequential language informs a methodology to make a language sequential, and hence suitable for sound REPL interpreters. The methodology enforces certain design principles on the REPL engineer to ensure that questions like the ones asked about the JShell interaction are answered precisely and are explicitly addressed as matters of language design, instead of an implementation concern. Furthermore, sequential languages are amenable to interfaces which allow *exploring* execution traces resulting from REPL interactions.

We have applied this methodology in three case studies. The first extends an existing implementation of MiniJava [11] in the Rascal language workbench [180], to make it sequential. This extended MiniJava is then the base interpreter for a computational notebook interface through Bacatá, Rascal's bridge to Jupyter [359]. The second case study involves QL, a DSL for defining spreadsheet-like interactive questionnaires [99, 100]. This case study show that it is feasible to obtain REPLs for languages that are not statement- or expression-oriented. The third case-study applies the methodology to obtain interactive services for eFLINT, a DSL for executable normative specifications [45].

The resulting services allow users and policy-aware software to navigate choices and decisions in the realm of law and regulation.

To summarize, the contributions of this chapter are:

- A feature-based analysis of the landscape of REPLs for a selection of the most popular programming languages (Section 3.2).

- A formalization of the notion of sequential language as the underlying principle of REPLs (Section 3.3).

- A language-parametric *exploring interpreter* algorithm on top of existing interpreters, allowing users to navigate user interaction history (Section 3.4).

- A methodology for developing REPL interpreters by *sequentializing* languages with a definitional interpreter (Section 3.5).

- Three case studies to illustrate the feasibility of the approach (Section 3.6).

The chapter is concluded with a discussion of limitations, related work, and directions for further research.

3.2 REPL DOMAIN ANALYSIS

Table 3.1: Surveyed REPL implementations.

REPL	Reference
CLing (C/C++)	https://cdn.rawgit.com/root-project/cling/master/www/index.html
JShell (Java)	http://openjdk.java.net/jeps/222
Python	https://docs.python.org/3/tutorial/interpreter.html
C#	https://www.mono-project.com/docs/tools+libraries/tools/repl/
Node.js (Javascript)	https://nodejs.org/api/repl.html
PHP	https://www.php.net/manual/en/features.commandline.interactive.php
PsySH (PHP)	https://psysh.org/
SQLite (SQL)	https://sqlite.org/
R	https://www.r-project.org/
Swift	https://swift.org/lldb/
Gore (Go)	https://github.com/motemen/gore
GNU Octave	https://www.gnu.org/software/octave/
Rappel (assembly)	https://github.com/yrp604/rappel
iRB (Ruby)	https://github.com/ruby/irb

This section provides a study of existing REPL interpreters and their main features. We have studied freely available REPL implementations, listed in Table 3.1, for the 15 most popular languages from the TIOBE index [61], with the exception of Visual
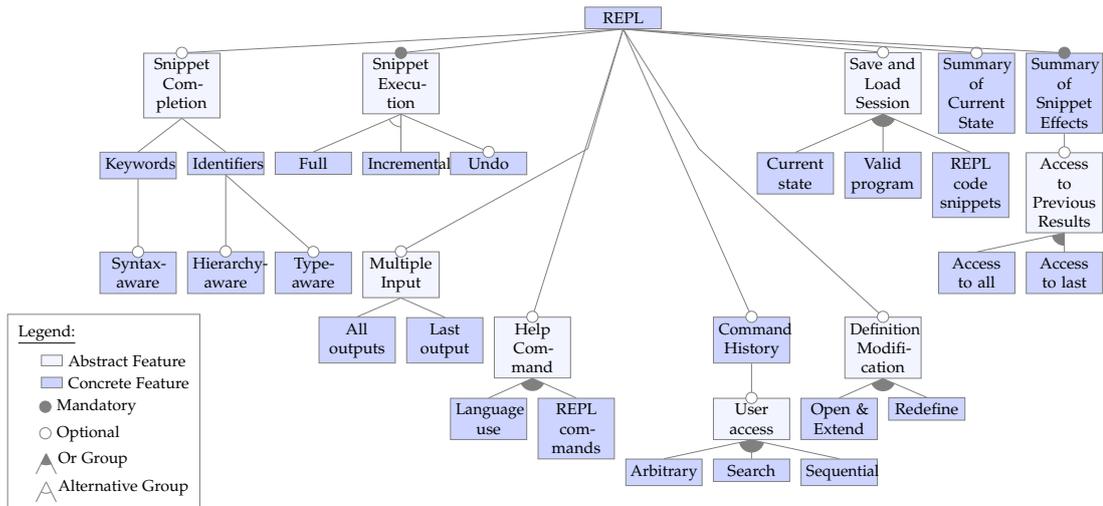
Figure 3.1: Feature model for REPL interpreters.

Basic, for which we could not find an freely available implementation. For MATLAB we have selected GNU Octave as a substitute. We performed a feature-oriented domain analysis [163], resulting in the feature model of Figure 3.1. Below we briefly describe the main mandatory and optional features.

MANDATORY FEATURES   An interpreter must have certain features to be considered a REPL. In particular, a REPL has the ability to execute multiple code snippets across multiple interactions in a single session (as opposed to executing one full program per session). In most of the investigated REPL implementations, the REPL maintains execution context and executes snippets incrementally (the "Incremental" alternative of the "Snippet Execution" feature). Optionally, a REPL may provide a way to undo the execution of snippets (roll-back). An alternative to incremental execution is composing all snippets into a single program and execute the program from scratch (the "Full" alternative). REPLs are expected to provide feedback after evaluating snippets, showing at least the snippet's printed output, and perhaps any result values or newly declared types ("Summary of Snippet Effects").

OPTIONAL FEATURES   Next to these mandatory features, the investigated REPLs implement several additional features such as auto-completion of snippets ("Snippet Completion"). This can target either language keywords or previously defined identifiers. Completion can take into account the syntactic context in which the user is typing, can be extended to fully qualified identifiers, and may also take into account the type of identifiers (through static typing or type hinting).

Even though the language itself might not support modifying an existing definition ("Definition Modification"), most REPLs allow this behavior to some extent. Common ways include overriding the previous definition, either through a new definition snippet or by editing it from an external text editor. Other REPLs also allow opening up definitions (such as classes) for additions ("Open & Extend").

Another common feature is the help (meta-)command ("Help Command"), which can document either the language, the REPL and its meta-commands, or both. The history of commands (including snippets) is usually made available to the user, in order to find and resubmit previous commands ("Command History"). It can be consulted sequentially through the arrow keys, but often includes a search facility as well. Some REPLs assign identifiers to commands in order to retrieve them arbitrarily. Some REPLs support saving and loading sessions ("Save and Load Session"). This may involve storing the execution context, or simply storing all user inputs to reproduce the execution context after loading. For some languages, the session can also be saved as a valid program outside of the REPL.

REPLs behave differently when multiple code snippets are input at once ("Multiple Input"). Output is either provided for all of the snippets or only for the last snippet (which could result in no output at all). Most REPLs allow inspection of the current execution context to the user ("Summary of Current State"). And finally, some REPLs allow the results of previous snippets to be used in new snippets ("Access to Previous Results"), either for the last executed snippet or for all by, for instance, assigning result values to variables.

FEATURE SUPPORT OF EXISTING REPLS   Table 3.2 shows how the investigated REPLs support the features identified in the feature model of Figure 3.1. The table illustrates that no two REPLs share the same set of features. IPython supports most features, whereas PHP supports a minimal set of features. Interestingly, PHP is the only REPL that does not print computed output values. The Go REPL (Gore) is the only REPL that simulates incremental execution by compiling a complete compilation unit in the back-ground. Type-aware completion is not applicable to Node.js and R since the languages are dynamically typed and do not support type hinting. Sessions exported from SQLite and R include the snippets to reproduce data, but not the results of querying the data. Octave exports variables and their values, but not declared methods. Only three REPLs support exporting sessions as valid programs. Although IPython provides additional commands, they are all implemented in Python and can therefore be exported. As explained before, a valid Go program is produced as part of every interaction with Gore. The interactive interpreter for Swift also provides debugging facilities. This feature was observed but not discussed as a REPL feature because the behaviors are accessed by running the interpreter in different 'modes'. Interestingly, the decision to provide both modes in a single tool was made from observing that the

Table 3.2: REPL Interpreter Features (●= full, ◓ = partial, − = not applicable).

		Cling	JShell	Python	IPython	C# REPL	Node.js	PHP	PsySH	SQLite	R	Swift	Gore	Octave	Rappel	iRB
Snippet Execution	Incremental	●	●	●	●	●	●	●	●	●	●	●		●	●	●
	Full												●			
	Undo	●														
Summary of Current State		●	●		●	●			●	●				●	●	
Summary of Snippet Effects		●	●	●	●	●	●		●	●	●	●		●	●	●
Access to Previous Results	Access to last				●	●	●		●					●		●
	Access to all		●		●							●				
Multiple Input	Last output	●			●	●			●			●		●	●	●
	All outputs		●	●						●	●					
Snippet Completion	Keywords	●		●	●		●		●	●	●	●		●		●
	Syntax-aware	●			●							●				
	Identifiers	●	●	●	●	●	●	●	●	●	●	●		●		●
	Type-aware		●			−						−			−	
	Hierarchy-aware	●	●	●	●	●	●		●			●	●	●		−
Definition Modification	Redefine		●	●	●	●					●	●	●	●	−	●
	Open & Extend														−	●
Help Command	REPL commands	●	●	●	●	●	●		●	●		●		●	●	
	Language use				●	●						●	●	●		
Command History (User Access)	Sequential	●	●	●	●		●	●	●	●	●	●	●	●	●	●
	Search	●	●	●	●	●	●	●	●	●	●		●	●		●
	Arbitrary		●		●									●		
Save and Load Session	Current state									●	●			◓		
	REPL code snippets		●		●		●			◓	◓					
	Valid programs				●		●			◓	◓			●		

The previous definition can be opened in an external editor for editing

modes shared similar features such as expression evaluation, data monitoring and step by step execution.

The wealth of features and diversity observed in REPLs motivated this chapter's study into the foundations of REPLs.

3.3 SEQUENTIAL LANGUAGES

This section defines the class of software languages for which the semantics can be expressed as a deterministic transition relation (a transition function). A subclass of these languages – the so-called *sequential languages* – is defined as the set of languages in which programs are written as sequences of smaller programs. A language is defined as a set of syntactically valid programs[3] with an interpreter assigning to each program the *effect* of the program, expressed as mutations on the context in which it is executed. The context is called a *configuration* in reference to Plotkin's Structural Operational

---

3 The abstract syntax of the language.

Semantics [272]. A program's effect is thus modeled as a function from configuration to configuration[4]. This model is sufficient to describe the semantics of real-world, large-scale, deterministic programming languages as is demonstrated by the body of literature on big-step, small-step and natural semantics [15, 162, 239, 247, 272] and does not exclude languages with non-deterministic aspects when these aspects can be captured algebraically [372].

Definition 3.3.1. A language L is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with P a set of programs, Γ a set of configurations, $\gamma^0 \in \Gamma$ an initial configuration and I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

Definition 3.3.2. A language $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator ; such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1; p_2 \in P$ and that $I_{p_1;p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.

Any two programs of a sequential language can be combined to form a new program whose effects are equal to the composition of the effects of the individual programs. If a language L does not have an operator ; with $I_{p_1;p_2} = I_{p_2} \circ I_{p_1}$, then the language is easily extended to have such an operator by taking $I_{p_1;p_2} = I_{p_2} \circ I_{p_1}$ as the definition of its semantics.

If the set P of the definition of a sequential language L is taken as the set of code snippets accepted by the REPL for L, then the ; operator describes the (snippet-related) behavior of the REPL. This is under the assumption that REPLs should always accumulate the effects of the code snippets they are asked to execute. The definitional interpreter of the language determines the effects of individual code snippets as well as the effect of their compositions. These observations show that, when it comes to the syntax and semantics of code snippets, REPL engineering can be considered as a matter of language design and engineering. On top of this there are several benefits to basing a REPL on a sequential language, i.e. to having the sequential composition operator as a language construct. These benefits are discussed throughout this chapter.

The ; operator of the Definition 3.3.2 does not necessarily correspond to the sequence operator of imperative or statement-based languages (often written as a semicolon). This is best exemplified by errors and exceptions. A sequence of statements typically terminates upon the occurrence of an exception, whereas a REPL is not expected to terminate if a code snippet raising an exception is submitted. As an example, consider

---

4 Corresponding to a big-step transition relation or the transitive closure of a small-step transition relation.

the following JShell interaction:

```
jshell> (1/0); System.out.println(5)
|  Exception java.lang.ArithmeticException: / by zero
|        at (#1:1)
jshell> (1/0)
|  Exception java.lang.ArithmeticException: / by zero
|        at (#2:1)
jshell> System.out.println(5)
5
```

This example shows that the effect of executing a snippet throwing an exception is to output information about the exception and to ensure that control flow returns to normal. In the first snippet, the print-statement is not executed, because of the exception. But after the second snippet, the user can continue printing a value.

Definition 3.3.3. Given a language $L = \langle P, \Gamma, \gamma^0, I \rangle$, the *reachability graph* from $\gamma \in \Gamma$ is the graph $\langle V, E \rangle$ with V and E the smallest sets of nodes and labeled edges such that $\gamma \in V$ and for every triple $\langle \gamma_1, p, \gamma_2 \rangle$, with $\gamma_1 \in V$ and $\gamma_2 = I_p(\gamma_1)$, it holds that $\gamma_2 \in V$ and that $\langle \gamma_1, p, \gamma_2 \rangle \in E$.

The reachability graph encodes, as paths, every possible execution run resulting from executing some sequence of programs in the context of configuration γ.

Lemma 1. The reachability graph from any configuration in a sequential language is closed under transitivity, with $\langle \gamma_1, p_1; p_2, \gamma_3 \rangle \in E$ if and only if there exists a $\gamma_2 \in \Gamma$ with $\langle \gamma_1, p_1, \gamma_2 \rangle \in E$ and $\langle \gamma_2, p_2, \gamma_3 \rangle \in E$.

Proof. Follows from the definitions of sequential languages and reachability graphs. $\square$

The effect of a program can be defined as the difference between the source and target configuration of an edge in a reachability graph, i.e. if $\langle \gamma, p, \gamma' \rangle$ is in some reachability graph, the effect of p in γ is the difference between γ' and γ. Lemma 1 states that the reachability graph for a sequential language is closed under transitivity. It follows that the effects of a *path* in the reachability graph, i.e. the effects of the sequence of programs occurring as labels on the edges of that path, is simply the difference between the start and end configuration of the path. The effects of a path can thus be computed without the need to compute the effects of all individual programs separately. Moreover, every path describes a valid program that can be saved (possibly together with its effects). Sequential languages thus admit simple implementations of the "Save and Load Session" feature of REPLs.

3.4   EXPLORING INTERPRETERS

This section defines a generic algorithm for executing programs by calling an underlying definitional interpreter and recording the resulting configurations. Specialized to a

particular language $\mathcal{L}$, this algorithm is the *exploring interpreter* for $\mathcal{L}$. The exploring interpreter for $\mathcal{L}$ records configurations in a subgraph of the reachability graph for $\mathcal{L}$ and is capable of reverting to a recorded configuration. Exploring interpreters admit exploratory programming by enabling programmers to revert to previous execution states in order to explore the effects of alternative sequences of code snippets.

The formal definition of exploring interpreters follows naturally from the definitional interpreter component of the definition of languages. However, definitional interpreters that form the basis for exploring interpreters have much stronger implementation requirements than definitional interpreters for REPLs without exploration. In particular, exploring interpreters require all program effects to be represented by changes in data, i.e. the definitional interpreter has to be a pure function. Without exploration, REPL implementations can be based on definitional interpreters that use real, rather than simulated, IO, memory and network communication (while still respecting their formal definition). Although exploring interpreters form the basis of the MiniJava and eFLINT case studies in this chapter, the core of the methodology we propose in Section 3.5 is also applicable to implementations without explicit state representation (as discussed in Section 3.5.1).

Definition 3.4.1. An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm maintaining a current configuration (initially γ^0) and an *execution graph* (initially containing just the node γ^0) and iteratively executing one of the following actions. At any moment the execution graph is a subgraph of the reachability graph from γ^0.

- **execute**(p): transition from the current configuration γ to the configuration $\gamma' = I_p(\gamma)$, where $p \in P$ is provided as input, and subsequently:
    - add γ' to the set of nodes (if new),
    - add $\langle \gamma, p, \gamma' \rangle$ to the set of edges (if new),
- **revert**(γ): take γ as the current configuration for the next action, where $\gamma \in \Gamma$ is provided as input.
- **display**: produce a structured representation of the current graph, distinguishing the current configuration in the graph from the other configurations.

The exploring interpreter algorithm is generic in that it has the language components P, Γ, I and γ^0 as inputs (type parameters in implementations). The **display** action can be used by interfaces to visualize the execution graph and to enable users to choose a node to **revert** to (see the MiniJava notebook discussed in Section 3.6).

REPL interpreters typically do not support the kind of exploration enabled by the **revert** action (only CLing has the "Undo" feature of the feature model). For practical purposes, such as space-efficiency, it may be desirable to implement a less powerful version of the algorithm in which the execution graph is maintained as a tree, a single

path or even just a single node. For example, an implementation of **revert**(γ) that removes all descendants of γ requires less space as it maintains only a single path (the exploring interpreter behaves like a stack). For another example, if a **execute**(p) is implemented to always create a new node for every γ', then the execution graph is actually a tree in which multiple nodes may hold the same configuration. There are advantages to both execution graphs (with sharing) and execution trees (without sharing). With sharing, there is the potential to avoid redoing (potentially costly) computations. This situation arises if the current node with configuration γ has an outgoing edge labeled q. If in this situation the action **execute**(q) is performed, then there is no need to interpret q as there is already an edge $\langle \gamma, q, \gamma' \rangle$ in the graph (for some γ'). The potential to avoid costly computations significantly increases if the graph is kept closed under transitivity (which is possibly for sequential languages according to Lemma 1) and if program transformations are used to label the edges with normal forms. Without sharing, there is exactly one path from the root node to every other node, i.e. every node has a unique 'history'. By reverting to a particular node, the user has not only chosen a configuration, but also the sequence of programs that led to that configuration. This is helpful, for example, when printing the effects of the snippets that gave rise to the current configuration after a **revert**. If, after reverting, the current node has multiple incoming edges, then it is not clear what output should be printed.

3.5 METHODOLOGY

In this section we propose a methodology for developing REPL interpreters based on the definitions and observations of the previous sections. The methodology proposes to build a REPL for some base language on top of an exploring interpreter for a sequential language defined as an extension of, or modification to, the base language. An exploring interpreter is essentially a bookkeeping device on top of a definitional interpreter and provides the "Incremental Snippet Execution" and "Undo" features directly (cf. Section 3.2). Additional motivation for using the exploring interpreter (for a sequential language) is that it promotes certain design principles while preserving the ability to implement many desirable features. These principles and their consequences are discussed in this section, together with a summary of the proposed methodology.

The core principles underlying our methodology are:

- the effects of a code snippet manifest as changes to an *explicit* state representation (a configuration)
- the effects of a code snippet are determined by the definitional interpreter used by the exploring interpreter
- the effects of a sequence of code snippets is the composition of the effects of the individual snippets
- only code snippets change configurations

For users of the REPL, the most important consequence of these principles it that an understanding of the definitional interpreter is enough to understand the precise behavior of the REPL for the language. In practical terms: to know the effects of code snippets, a user needs to understand the base language and its extension or modification to a sequential variant. The extension or modification is made explicit by the definitional interpreter and should be communicated clearly (as precise documentation, a formal semantics, or an open-source implementation).

For engineers of the REPL, the most important consequence of the principles is that every feature (on top of "Incremental Snippet Execution" and "Undo") is implemented either:

- as a language extension (e.g. the features "Definition Modification" and "Access to Previous Results"),

- as a series of interactions with the exploring interpreter (e.g. "Multiple Input", explained below),

- based on information stored in the execution graph (e.g. "Summary of Snippet Effects", "Summary of Current State" and "Snippet Completion") or

- independently of the exploring interpreter, when the feature does not involve snippet execution (e.g. "Help Command" and other meta-commands).

The methodology of this chapter is based on the hypothesis that many of the features of existing REPLs, including at least those in Figure 3.1, fall into the four categories listed above. This hypothesis is tentatively supported by the various feature implementations described across Section 3.6.

To prelude the example feature implementations of Section 3.6, consider the alternatives of the "Multiple Input" feature ("All outputs" and "Last output"). A "Multiple Input" snippet is parsed as, for example, $p; q; r$. In the implementation of a REPL following our methodology, such a snippet can be handled by performing three **execute** actions with respectively p, q and r as inputs (because the language is sequential). The REPL has then seen the four configurations γ_0, γ_p, γ_q and γ_r corresponding to the configuration before executing p, after executing p, after q and after r respectively. The output of the last input r is found by computing the difference between γ_q and γ_r, the output of all three inputs is found by computing the difference between γ_0 and γ_r.

The methodology for developing a REPL for any base language $\mathcal{L}$ is formulated as the following steps (and has certain commonalities with the approach of [152]).

1) DEFINITIONAL INTERPRETER   Formulate $\mathcal{L}$ as a language in terms of its concrete and abstract syntax, and a definitional interpreter that captures the effects of programs as a function over some set of configurations, thus forming the components of a language according to Definition 3.3.1. If the language is sequential according to Definition 3.3.2, then steps 2–5 can be skipped.

2) *phrase* NONTERMINAL   To define a sequential variant $\mathcal{L}'$ of $\mathcal{L}$, reuse the syntax definitions of the previous step to define a new sort **phrase** with an alternate for each of the sorts of $\mathcal{L}$ that describe the syntax of a valid code snippet of the envisioned REPL. The syntax can also have other extensions or modifications, as long as **phrase** is the entry point of the syntax (the first component of a language in Definition 3.3.1).

3) *phrase* INTERPRETER   Define a definitional interpreter for $\mathcal{L}'$ to capture the semantics of phrases, reusing as much as possible the definitional interpreter of step 1, ideally by applying modular extension mechanisms (e.g., Object Algebras [125, 260], Rascal's `extend` [28]). Special consideration needs to be given to the effects of phrases to ensure the next phrase is executed in the right context. For example, if the result value of a phrase needs to be available to the next phrase through a binding, this binding needs to be introduced as one of the effects of the first phrase.

4) ";"-PHRASE   Extend the sort **phrase** with an alternate that combines two valid phrases to form a phrase. For example, with the semicolon as a separator, let $p;q$ be a valid phrase if p and q are valid phrases.

5) INTERPRETER FOR ";"   Extend the definitional interpreter of $\mathcal{L}'$ such that the effect of a phrase formed by combining two phrases is the composition of the effects of the combined phrases, e.g. $I_{p;q} = I_q \circ I_p$. The language $\mathcal{L}'$ is sequential by definition as a result of this and the previous step.

6) INSTANTIATE EXPLORER   Obtain an exploring interpreter for $\mathcal{L}'$ by instantiating the generic exploring interpreter algorithm with the definitional interpreter for $\mathcal{L}'$. The implementation may be simplified compared to Definition 3.4.1 in that it maintains a simpler form of execution graph, if desirable. Instead of an exploring interpreter, the definitional interpreter for $\mathcal{L}'$ can also be used directly. In fact, any implementation that respects the semantics of the definitional interpreter can be used, e.g. an implementation with real rather than simulated effects.

The chosen interpreter can then be offered through various user interfaces, such as command-line interfaces, a network service, or a computational notebook. The interface displays visualizations of the effects of phrases, e.g., by showing output, computed values and new bindings, and can optionally implement additional REPL features.

3.5.1 *Pragmatics*

In the context of language workbenches [99] and DSLs [235], a common language implementation strategy is to define interpreters, consisting of functions traversing an abstract syntax tree whilst modifying a propagated configuration to express effects (following the Visitor design pattern). The case studies of the next section include such

interpreters. The REPLs in these case studies are obtained through generic implementations of the exploring interpreter algorithm (in Java and in Haskell) that are easily specialized by providing the entry points of the abstract syntax and the interpreter. The presented methodology is based on an exploring interpreter because it is a relatively natural and simple layer to add on top of the described definitional interpreters typically built with Rascal [28, 180]. Moreover, the generic exploring interpreter forms a suitable abstraction for reasoning about sequences of interactions between programmer and REPL – e.g. saving and loading sessions and extracting base language programs – and for implementing advanced REPL and notebook features that support exploratory programming and live programming.

In theory, our approach can also be used for developing REPLs for (general-purpose) programming languages, as many languages can have their semantics expressed as a transition function. In practice, however, very few programming languages have an interpreter implemented as a pure function or have a complete operational semantics from which such an interpreter can be derived. REPLs are not typically implemented with explicit state representation and few enable backtracking (in our survey only CLing supports "Undo"). However, an impure interpreter implementation can be used at step 6 (Instantiate Explorer) of the methodology. Although some advanced features – such as "Undo" – may then be harder to realize, the most important principles of our methodology still hold. In particular, the differences between base language and REPL should be formulated as extensions or modifications of the base language. This is achieved by updating the semantics of the base language such that repeated execution of its interpreter (i.e. the composition of effects) gives the behavior expected of the REPL of the language. The details of how this can be achieved depend on the language and the techniques used to implement the language. Discussed next are the general patterns that have been observed in our survey.

3.5.2 *Common REPL Language Extensions*

As mentioned in Section 3.3, languages rarely provide an operator that corresponds precisely to the REPL top level. For example, a snippet with an uncaught exception is not expected to prevent subsequent snippets from being executed, whereas termination is expected when an exception occurs within a sequence of (;-separated) statements. Of the surveyed REPLs, only Gore prevents subsequent snippets from executing once a previous snippet raises an exception (a consequence of its "Full" execution model). In the other languages, the REPL top level catches any otherwise uncaught exceptions and presents them to the programmer after which a subsequent snippet can be executed. In languages with constructs for catching and handling exceptions, one might explain or implement this feature with a top-level catch and a handler that prints the exception. For example, a snippet `{System.out.println(1); (1/0);}` can be considered as implicitly wrapped in a **try**/**catch** block in JShell as follows:

```
try {{System.out.println(1); (1/0);}} catch (Exception e) {
  ... // print the exception in a helpful format
}
```

This clarifies, in reference to the Java semantics, that any effects produced by a snippet before, but not after, an exception is thrown are preserved. However, the translation is inaccurate as a JShell snippet is not an isolated block, unlike a `try`-block. Bindings produced by top-level declarations are active when subsequent snippets are executed, i.e. all snippets are in the same scope and the top-level catching exceptions does not change this. In the next JShell fragment, the meta-variable `$1` is available to subsequent snippets despite the exception.

```
jshell> 5; (1/0);
$1 ⇒ 5
| Exception java.lang.ArithmeticException: / by zero
|       at (#2:1)
```

This example also highlights the importance of presenting new bindings, assignments, and any other effects to the programmer, providing the information required by the programmer to update their mental model of the REPL's execution state.

Another common example of a modification to the base language is the "Access to Previous Results" feature available in several REPLs of the survey (demonstrated by the variable `$1` in the above fragment). JShell and IPython ("Access to All") implement this feature as follows. Whenever a code snippet produces a result value (other than void), this result value is assigned to a fresh variable. For example, if the second snippet sent to IPython produces result value 5, then the variable `_2` is assigned 5. The behavior differs between JShell and IPython when a code snippet contains multiple statements. In IPython ("Last Output"), the result of a sequence of statements is the result of the last statement[5], e.g., the snippet `print(1);2;print(3)` prints 1 and 3 but has no result value. In JShell, the result of a sequence of statements is the result of each statement with a (non-void) result. If a snippet has multiple results, each result is assigned to a fresh variable. For example, if `3;2;System.out.println(1);` is sent as the first snippet to JShell, then the variables `$1` and `$2` are assigned the values 3 and 2 respectively and 1 is printed. In Node.js ("Access to Last"), a statement such as `console.log(1)` produces `undefined` as a result, which is then assigned to the variable `_`. PsySH also assigns the last uncaught exception to the variable `$_e`. This feature is helpful in situations where the exception is not easily reproduced, e.g., when caused by a (rare) non-deterministic, pseudorandom or timed event.

Most languages of the survey enable definitions to be redone ("Definition Modification"), with only iRB also allowing extensions to existing definitions ("Open & Extend"). The main challenge to redefining or modifying existing definitions is checking whether an updated definition is consistent with definitions that depend on it. This

---

5 Even when void. A possible alternative is to use the last non-void result.

is particularly challenging for statically typed languages such as Java. In JShell, any inconsistencies are reported when a (now incorrect) definition is used, as shown by the following interaction:

```
jshell> class B {int mymethod(){return 0;}}
|  created class B
jshell> class A {int mymethod(){return new B().mymethod();}}
|  created class A
jshell> class B {long mymethod(){return 0;}}
|  replaced class B
jshell> int x = 4; int y = new A().mymethod(); int y = 5;
x ⇒ 4
|  attempted to use class A which cannot be instantiated or
|  its methods invoked until this error is corrected:
|   possible lossy conversion from long to int
|   class A { int mymethod() { return new B().mymethod(); }}
y ⇒ 5
```

Note that the last snippet is neither type-checked and rejected as a whole nor that the error keeps the other statements from being executed. Statements appear to be type-checked individually, with any errors causing only the individual statement to be rejected. However, the following JShell interaction shows that this is a simplification:

```
jshell> int x = 1; new A(); int y = 2;
x ⇒ 1
|  Error:
|  cannot find symbol
|    symbol: class A
```

A downside of showing inconsistencies just before they cause problems is that a menial mistake can cause a cascade of avoidable mistakes to go undetected, perhaps requiring tedious efforts to resolve. A downside of reporting inconsistencies as soon as they arrive is that they may be considered redundant and a nuisance when a programmer is aware and about to resolve the inconsistencies.

The C# REPL does not update method definitions affected by an update to another class. So when, in the example above, mymethod is called on a new instance of A, the behavior is that of the old mymethod of class B. (A similar example using fields rather than methods causes the C# REPL to hang.)

A general theme in the discussed language extensions is that they relate to the effects of code snippets on their successors. A REPL engineer should consider all the different kinds of (side-)effects code snippets can produce and decide for each effect whether it should propagate and, if so, how the programmer is informed of the effect, enabling them to update their mental model of the REPL's state. To help the programmer further, the ability to request an overview of the currently active bindings is desirable, especially together with a mechanism for inspecting (modified) type definitions.

3.6   CASE STUDIES

This section discusses several REPL implementations for a number of languages with different user interfaces. The section is structured according to three case studies for the Rascal-defined languages MiniJava and QL, and the Haskell-defined language eFLINT. The case studies implement novel sequential variants of these languages.

3.6.1   *A Jupyter Notebook for MiniJava*

The MiniJava language is a subset of Java that retains the essential object-oriented features of Java [11, 64]. The semantics of a MiniJava program is given by its interpretation as a Java program. It is implemented as a definitional interpreter in the Rascal language workbench [180]. The extension to a sequential MiniJava uses Rascal's modular extension mechanisms and demonstrates the methodology of the previous section.

The first part of the extension is choosing the top-level constructs of the language. As for **JShell**, these are expressions, statements, variable, class, and method declarations, and their associative composition. The syntax of MiniJava is extended by adding the `Phrase` construct:

```
syntax Phrase
 = Expression ";" | Statement
 | VarDecl | ClassDecl | MethodDecl
 | assoc Phrase Phrase;
syntax Statement
 = ...
 | "throw" "new" StringLiteral ";";
syntax Expression
 = ...
 | Identifier "(" ExpressionList? ")";
```

The extension also includes a new method call variant, enabling (global) methods to be called without a receiver. The **throw**-keyword is added to demonstrate an implementation of handling uncaught exceptions. Exception values are simplified to string literals rather than than arbitrary objects.

The definitional interpreter of extended MiniJava is defined in Rascal as the function `Config eval(Phrase, Config)`, shown[6] in Listing 3.1. The type `Config`, shared by both

---

6 The notation `(NT)'...'` is used to pattern match against or construct concrete syntax trees of type `NT`, where `NT` is some nonterminal defined in Rascal's native grammar formalism; the parts between fish-angle brackets represent typed holes of the pattern.

MiniJava interpreters, is defined as the following tuple type:

```
alias Config = tuple[
  Env env, Sto sto,
  int seed, Out out,
  Val given, MaybeFailure failed,
  Val result
];
data MaybeFailure
  = failure(FailureType e)
  | no_failure()
  ;
data FailureType
  = failed()
  | exception(str msg)
  ;
```

Configurations have the following fields: the current execution environment (`env`), the store (`sto`), a seed (`seed`), the output of all executed phrases represented as a list of strings[7] (`out`), a given value (`given`) of type `Val` used for passing arguments, the field `failed` to indicate if and why the execution got 'stuck', and a value with the execution's result (`result`). The `Val` Algebraic Data Type (ADT) (not shown) defines constructors for references, integers, booleans, vectors (arrays), environments, lists, closures, classes, objects, and `null`. The alternative `failed()` of `FailureType` indicates the execution got stuck because the evaluated program is invalid (e.g. due to unbound variables). The alternative `exception(str msg)` indicates an exception has been thrown with exception value `msg`.

The cases of Listing 3.1 that handle declarations (class, variable, or method) first produce an environment by calling the respective functions `declareClass`, `declareVariables` and `declareGlobalMethod`. These functions also produce output that informs the programmer of the successful binding of the respective class, variable or method. If a class is redefined, the programmer is also informed. The `collectBindings` function (not shown) adds the bindings in the computed environment (`result`) to the execution environment (`env`). The function `catchExceptions` (not shown) checks whether a phrase has failed or raised an exception. If so, the failure or exception is reported and removed, ensuring that the next phrase executes normally. Note that a MiniJava code snippet of the form `1;(2/0);3;` is parsed as a sequence of three phrases and not a code block consisting of three statements. Since the division by zero error is removed, the next phrase (`3;`) is executed normally. So, contrary to JShell, there is no distinction between phrases executed as separate code snippets or as a single, semi-colon separated code snippet. This arguably makes the language more consistent. The behavior of statements separated by a semi-colon in code blocks is unaf-

---

7 The implementation converts the integers printed by MiniJava to strings and inserts a newline, corresponding to Java semantics.

fected and an exception will terminate the execution of a code block when it arises.

Listing 3.1: Interpreting MiniJava phrases.

```
Config eval((Phrase)'<Expression e> ;', Config c)
 = catchExceptions(collectBindings(
    setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)'<Statement s>', Config c)
 = catchExceptions(collectBindings(
    setOutput(exec(s, c))));

Config eval((Phrase)'<ClassDecl cd>', Config c)
 = catchExceptions(collectBindings(
    declareClass(cd, c)));

Config eval((Phrase)'<VarDecl vd>', Config c)
 = catchExceptions(collectBindings(
    declareVariables(vd, c)));

Config eval((Phrase)'<MethodDecl md>', Config c)
 = catchExceptions(collectBindings(
    declareGlobalMethod(md, c)));

Config eval((Phrase)'<Phrase p1> <Phrase p2>', Config c)
 = eval(p2, eval(p1, c));
```

The first two cases of Listing 3.1 deal with expression and statement phrases, reusing the original interpreters for expressions and statements (eval and exec respectively). A statement, which may be a code block consisting of multiple statements, either computes null or an environment that contains the bindings for all variables that have been assigned a (new) value. The function setOutput (not shown) inspects the computed bindings, if any, and prints the variable and its assigned value, matching the behavior of JShell. An expression computes a value such as an integer, a boolean or an object reference. The function createBinding (not shown) assigns the computed value to a fresh variable, using the seed field of the current configuration, and binds the fresh variable to the identifier $<i>, where <i> is generated from the seed. The applications of setOutput and collectBindings ensure that the new binding is reported to the programmer and is active when the next phrase is executed, matching the behavior of JShell.

The final case confirms that two consecutive phrases are evaluated by function-composition. The implementation of method calls without receiver expression is not given.

The definitional interpreter of the extended language forms the interface to language services such as REPLs and computational notebooks. The connection between the definitional interpreter and Rascal's notebook framework Bacatá is discussed next.

EXPLORING INTERPRETERS IN BACATÁ   Bacatá [359] is a generic Jupyter [161] ker-
nel generator for languages developed within the Rascal Language Workbench. Bacatá
is extended to support notebooks based on exploring interpreters. The generic imple-
mentation of the exploring interpreter maintains a full execution graph (in accordance
to Definition 3.4.1). Bacatá relies on the definition of a language `repl`, a value of the REPL
ADT shown below:

```
data REPL[&T]
= repl(&T initConfig, &T (str, &T) handler,
  Completion (str, int, &T) completor, Content (&T, &T) printer);
```

A value of REPL contains all required information to build a REPL command-line
interface for a language, or, together with Bacatá, a computational notebook. The type
parameter `&T` represents the configuration (e.g., `Config` of MiniJava). The `handler` takes a
line of input and a configuration and produces a new configuration. The completor can
be provided for tab-completion services. Finally, the `printer` produces (HTML) content
from the previous and/or current configuration.

Bacatá is used as an interface between a Jupyter server and the language's REPL.
The workflow that describes the communication among these components is as follows:
Jupyter takes the user's code snippets and sends them to the language's interpreter
through Bacatá. Bacatá takes the user's code and calls the language's `handler` (defined
in the `repl` value), which is responsible for calling the parser and then the interpreter of
the language. Finally, the `handler` produces a result, which is then displayed to the user,
using the `printer`.

Figure 3.2 shows a simple notebook for MiniJava, produced with Bacatá. The right
shows the execution graph for exploring the user's interaction with the notebook. The
edges of the graph are labeled with the corresponding cell number, and the node
representing the current configuration is highlighted in green. The user can click any
other node to make it active. The following cell will then be executed in the context of
that exact configuration, resulting in a split in the graph if the resulting configuration
differs from the activated one.

A NOTEBOOK INTERFACE FOR MINIJAVA   Obtaining a REPL-style command-line
or notebook interface for MiniJava amounts to instantiating the REPL data type with
the appropriate handlers, printers, and completors. In the case of a Bacatá-generated
notebook Jupyter interface, the programmer has access to a visual representation of the
execution graph of the exploring interpreter, as shown in Figure 3.2.

The handler for MiniJava parses the incoming input as a `Phrase` and calls the extended
definitional interpreter, which returns a new configuration. The printer takes the old
and new configuration and prints relevant output. After a successful execution, the
differences between the `out` components of the new and old configuration is shown.
In the case of a declaration, the difference between the two `env` components gives the
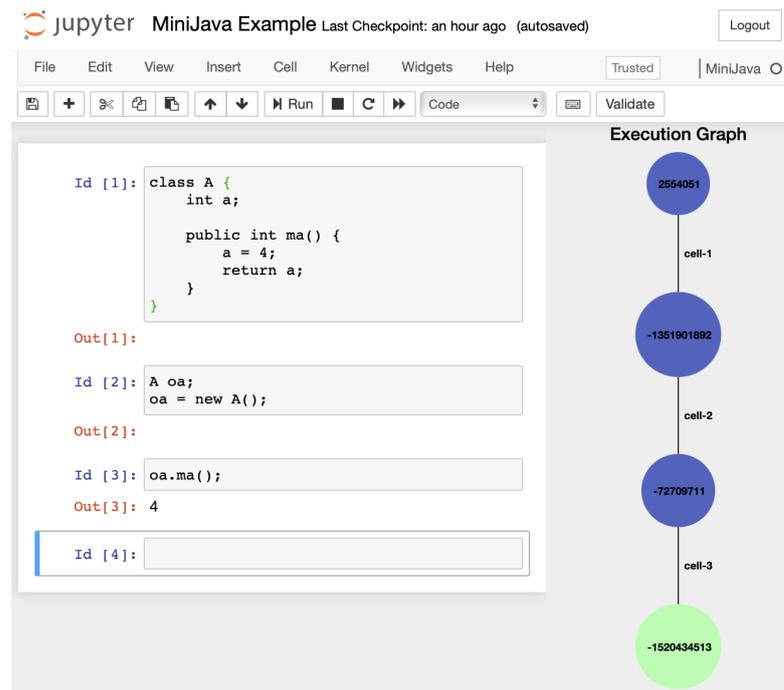
Figure 3.2: Notebook example.

new bindings. The completor uses the bindings in its input configuration to suggest possible completions for identifiers.

3.6.2   *QL: A DSL for Questionnaires*

QL is a little language for defining interactive questionnaires [99, 100], like tax filing forms or online surveys. A QL form defines a sequence of questions, where each question has a label, an identifier, a type (boolean, integer, or string), and an optional expression if the question is computed. Expressions contain the usual arithmetic and comparison operations, and allow referring to the current value of another question. Furthermore, questions can be made conditional using if-then and if-then-else constructs.

The meaning of a QL program is a rendering as an interactive GUI program, where the user enters values for the (non-computed) questions. Depending on this input, conditional questions may be shown or hidden, and the value of computed questions may be recomputed, similar to a spreadsheet. A simple example is shown in Figure 3.3, including its rendering as an interactive UI.

From a REPL perspective, QL is interesting, because a form specifies a conditional data-flow network rather than a program consisting of instructions. Nevertheless, in

```
form taxOfficeExample {
  "Did you sell a house in 2010?"
    hasSoldHouse: boolean
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean
  "Did you enter a loan?"
    hasMaintLoan: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: integer
    "Private debts for the sold house:"
      privateDebt: integer
    "Value residue:"
      valueResidue: integer =
        sellingPrice - privateDebt
  }
}
```

Did you sell a house in 2010? ☑
Did you buy a house in 2010? ☐
Did you enter a loan? ☐
What was the selling price?
Private debts for the sold house:
Value residue:
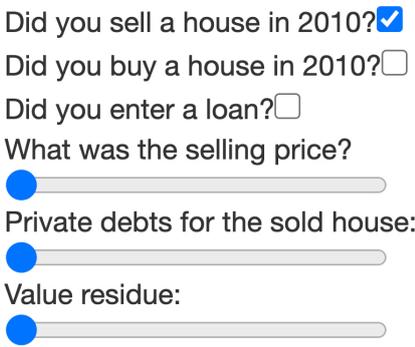
Figure 3.3: QL questionnaire and its rendering.
A QL questionnaire (left) and its rendering (right).

this section we introduce a prototype REPL for QL, both as an instructive thought experiment, and to stress the concept of sequential language.

Abstractly, the semantics of QL can be described with the following (Rascal) function signature:

```
tuple[UI, Env] eval(Form form, Env env, Event evt);
```

Given a form, an environment mapping question identifiers to values (Env), and a user event (Event), the function eval produces a rendering (UI) and an updated environment. Running a QL questionnaire then amounts to constructing an initial rendering, and then updating the current environment and redrawing the UI after every user action.

To provide a REPL interface for QL, we extend the language with a new start nonterminal, Cmd, the definition of which is shown in Listing 3.2. Commands are the snippets that the user can enter at the command line.

Listing 3.2: Language extension for ReplizedQL.

```
syntax Cmd
  = Form            // define form
  | Question        // append a question
  | Question "..."  // prepend question
  | "@" Addr Question // replace question
  | Expr            // evaluate expression
  | Id "=" Value;   // perform user action

syntax Script
  = Cmd* commands  // batch perform commands
```

The first four alternatives of `Cmd` capture constructs to manipulate forms. The user can define complete forms, append or prepend individual questions to the current form, and replace questions arbitrarily nested in the form using a positional reference mechanism (`Addr`).

The last two alternatives can be used to evaluate expressions, which shows the result, or update the value of a (non-computed) question, if the current state of the UI allows it. The update-value action simulates a user interaction if the form would have been rendered as a proper UI. Finally, Listing 3.2 defines a `Script` nonterminal to combine multiple commands in sequence.

The interpreter for commands is a function from a command and the current configuration to a new configuration:

```
Config eval(Cmd cmd, Config cfg) { ... }
```

The `Config` type captures the current environment, the current form, and a list of output values (UI renderings and expression evaluation results).

That our definition of QL is sequential can be seen from the definition of the interpreter for `Script`s:

```
Config eval(Script scr, Config cfg) =
  ( cfg | eval(cmd, it) | Cmd cmd ← scr.commands );
```

This function simply composes the `eval` function for commands for every command in the script[8]. This follows the definition of sequential language of Section 3.3.

A SAMPLE INTERACTION   The above interpreter for commands can be hooked to Rascal's standard REPL infrastructure to obtain a command-line interface for QL. We illustrate the semantics of sequential QL below, using a sample user interaction. The code snippets use > as prompt, the output of a command is shown directly below.

First, let's define a simple form:

---

8 The notation (init | ... **it** ... | gen) is Rascal syntax for writing a reduce operation.

```
> form simple { }
  ·
```

The result is the empty rendering of the UI, indicated by ·. Then we append a (computed question), labeled "A", of type integer:

```
> "A" a: integer = c + b + 1
  A ·
```

The a question is not conditional, so it is shown in the UI rendering; note however that the value of the question is still undefined because questions c and b have not yet been defined.

The b question could be defined as follows:

```
> if (a < 20) "B" b: integer = c + 1
  A ·
```

Since b is still undefined (because c is), a remains undefined as well, and as a result, the visibility condition of b evaluates to false. This all changes, however, after defining c:

```
> if (a > 20) "C" c: integer
  A 2
  B 1
```

The question c is not computed, so it receives an initial default value (in this case 0). Both a and b can now be computed, as well as the condition of b, causing b to be shown in the UI. Now let's change the value of c:

```
> c = 10
  A 22
  C [10]
```

Setting c to 10 disables b, but changes the visibility condition of c to true, making it appear in the UI. The square brackets around the value of c indicate it is editable.

Changing the value of c to 5 updates the UI accordingly:

```
> c = 5
  A 12
  B 6
```

Now b becomes visible, and c is hidden again.

It is possible to add questions to the beginning of the form:

```
> "D" d: integer = 3 * a...
  D 36
  A 12
  B 6
```

Or using the path-based address notation:

```
> :form
form simple {
 [0] "D" d: integer = 3 * a
 [1] "A" a: integer = c + b + 1
 [2] if (a < 20)
  [2.0] "B" b: integer = c + 1
 [3] if (a > 20)
  [3.0] "C" c: integer
}
> @2.0 "c + 1 is:" b: integer = c + 1
 D 36
 A 12
 c + 1 is: 6
```

The `:form` meta-command pretty-prints the current form annotated with addresses for every question. Using the `@`-notation, the user can replace any question in the form, in this case to change the label of the `b` question.

Note that the append-, prepend-, and position-based adding and replacement of questions can be considered a rather low-level (maybe even pathological) way of editing a program (reminiscent of the line-based editors of the past). Nevertheless, without neccessarily claiming this is a realistic way of evolving programs, it does illustrate a kind of REPL "completeness", where every program and program change can be realized using commands at the prompt.

3.6.3 *eFLINT: Executable Normative Specifications*

eFLINT is a DSL for developing executable normative specifications used to reason about compliance with regulations, contracts and/or policies [45]. eFLINT programs are used to simulate or verify normative decision making processes. The methodology of Section 3.5 has been applied to develop two REPLs on top of one exploring interpreter for eFLINT. The implementation of eFLINT is available at GitLab [40].

repl interfaces   The first REPL is a command-line tool for exploring compliant and non-compliant behavior. Figure 3.4 shows an example session where the user explores the norm "children can ask their parents for help". As a meta-command, a user can choose actions and events to trigger from a given list of options. Choosing an action or event has the effect of updating a database of 'facts', representing the state of the world at a particular moment in time. A fact is said to 'hold true' if it is present in the database. Some facts are reifications of actions and correspond to acceptable behavior when they hold true and when they are enabled by their pre-conditions. Disabled actions can be executed in order to explore non-compliant behavior (although causing a violation). Other facts represent duties, which need to be 'terminated' before one of

```
#0 > Fact person. Placeholder parent,child For person
new fact-type person
no enabled actions or events
#3 > +person(Alice). +person(Bob) // introduce persons
+"Alice":person
+"Bob":person
no enabled actions or events
#5 > Fact parent-of Identified by parent * child
new fact-type parent-of
no enabled actions or events
#6 > +parent-of(Alice,Bob)
+("Alice":person,"Bob":person):parent-of
no enabled actions or events
#7 > Act call-for-help Actor child Recipient parent
                        Holds when parent-of()
new fact-type call-for-help
+("Bob":person,"Alice":person):call-for-help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#8 > :choose 1      // Bob asks Alice for help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#9 > :revert 7      // to before the action was declared
+("Alice":person,"Bob":person):parent-of
#7 > :current       // show the current set of facts
"Alice":person
"Bob":person
("Alice":person,"Bob":person):parent-of
#7 > ?Enabled(call-for-help(Bob,Alice)) // query
undeclared type: call-for-help
```

Figure 3.4: A session with the eFLINT command-line REPL.

their violation conditions holds true. The phrases of the language are declarations of fact-, act-, event- and duty-types, action or event triggers, insertion and removal of facts and queries on the database. After a phrase is executed, the user is presented with the changes in the database, newly defined types, any violations and a new list of options.

The second REPL is a TCP server that listens on a chosen port for incoming phrases and responds with the same information as the command-line REPL (in JSON form). The TCP server is used as a general method for connecting other languages with eFLINT to benefit from the normative specification written in eFLINT. For example, a program can send queries to the eFLINT server to check whether certain actions are enabled before actually performing them. In this way, software can be developed that is 'compliant by design'.

The REPLs are developed on top of an exploring interpreter for eFLINT, briefly explained next.

EXECUTION TREE   The type *Explorer* is an alias for functions that receive an *Instruction* and return a *Response* in the *IO* monad (Haskell's mechanism for input and output).

> **type** *Explorer*   = *Instruction* → *IO Response*
> **data** *Instruction* = *Execute CPhrase* | *Revert Int* | *Display*
> **data** *Response*   = *Success Node CPhrase Node* | *ExecError Error*
> **type** *Node*   = (*Int*, *Config*)

The values of *Instruction* correspond to the actions of the generic exploring interpreter algorithm. There are two types of response, for successful executions and failing executions respectively. One of the values of *Error* indicates that the integer given as part of some **revert** action does not correspond to a known configuration. The success response contains the elements of an edge in the execution graph: two nodes and a label (phrase). The edge gives the effects, in terms of an input and output configuration, of the last phrase executed by the exploring interpreter. A node is a configuration and an integer that uniquely identifies the node. The label is a value of type *CPhrase*, a phrase that has been compiled.

A configuration contains information about declared types (a type environment), a database of facts and a list of output holding any reported violations:

> **data** *Config* = *Cfg* { *tyenv* :: *TyEnv*, *state* :: *Set Fact*, *out* :: [*String*] }

The algorithm maintains a tree rather than a graph, and does so in a way that makes it very simple to find the path from the root to any given configuration in the tree. The type *SIDMap* is an alias for a map mapping integers to the configurations with which they form a node. The type *History* represents a tree as a collection of edges.

> **type** *SIDMap* = *IntMap Config*
> **type** *History* = *IntMap* (*Int*, *CPhrase*)

If x maps to (y, p) in the *History* map, this means that there is an edge $\langle \gamma, p, \gamma' \rangle$ in the tree where y is the integer identifying γ and x is the integer identifying γ'.

REPL FEATURES   The function *getPath* :: *Int* → *SIDMap* → *History* → [*CPhrase*] receives an integer identifying a node and uses the maps to compute the sequence of phrases labeling the path from the root of the tree to the node. The function is used to save a session by pretty-printing and storing the returned phrases in a file.

The definitional interpreter of eFLINT receives compiled phrases (*CPhrase*s) as input. The tool-set for eFLINT contains a compiler that translates from *Phrase* to *CPhrase*. The compiler checks whether a *Phrase* is well-typed and applies conversions to make explicit certain implicit operator applications. Compilation is performed by the function *compile* : *TypeEnv* → *Phrase* → *CPhrase*, receiving as input the type environment of the current configuration held by the exploring interpreter.

When the command-line or TCP server REPL receives a *String* for execution, the string is parsed as a *Phrase*. If successful, the *Phrase* is type-checked and compiled to a *CPhrase*. The *CPhrase* is sent as an **execute** action to the exploring interpreter, which invokes the definitional interpreter and responds either with an error or with the edge of its graph representing the latest execution. This edge is given to a function called *effectsOf* to compute the effects of executing the phrase. The function *effectsOf* finds any new bindings by computing the difference between the two type environments of the input configurations, finds any created or terminated facts by computing the difference between the two state components and finds new violations by computing the difference between the two output components.

3.7 DISCUSSION & RELATED WORK

LIMITATIONS & FUTURE WORK   The techniques described in this chapter are applicable to languages that can be implemented by deterministic interpreters with explicit state representations Moreover, if an execution graph is not needed, then state does not have to be represented explicitly (see Section 3.5.1), as long as the effects of top-level phrases still compose and are communicated clearly to REPL users. This requirement does not necessarily rule out concurrent, non-deterministic, compiled or data flow languages. In some cases it is possible to *model* the complicating aspects of these languages, e.g., with thread models, data flow graphs and lists to capture non-deterministic results.

Purely functional interpreters with explicit state representation are, however, further removed from actual implementations and may be less suitable for developing practical REPLs. For instance, a definitional interpreter for C can model memory (pointers) rather than providing real memory access. A REPL for C can also be based on an interpreter that invokes a C compiler, wrapping current and previous code snippets in `int main() {…}`, before compiling and executing the resulting program (similar to the Go REPL). It is possible to obtain a REPL interface in this way, but it would not be based on a sequential language and the explorative quality of exploring interpreters is lost. The applicability of our approach in the context of such compilation-based REPLs is to be investigated further.

The interpreters discussed in this chapter are all implemented in functional programming languages (Rascal and Haskell) with immutable data. Maintaining the execution graph is therefore easy to implement, but it may come at a cost of performance and memory footprint. Further research is needed to represent the graph more efficiently, for instance by maximizing sharing, caching intermediate results, or selectively culling the graph. The pragmatics of a REPL (small snippets, immediate feedback, etc.), however, suggest that such optimization might be premature.

Although not shown in this chapter, exploring interpreters can also be used to realize additional features not typically found in REPLs by performing sequences of **execute** and **revert** actions in response to a single user action. For example, if a user edits a cell

```
U:   Type 2+2.
J:          2+2 =        4

U:   Set x=3.
     Type x.
J:          x =          3
U:   Type x+2, x-2, 2·x, x/2, x*2.
J:          x+2 =        5
            x-2 =        1
            2·x =        6
            x/2 =        1.5
            x*2 =        9

U:   Type [(|x-5|·3+4)·2-15]·3+10.
J:   [(|x-5|·3+4)·2-15]·3+10 = 25
```

Figure 3.5: Early user interaction using JOSS.
Early user interaction using JOSS [316].

in a notebook, this could cause the exploring interpreter to revert to the configuration in which that cell was originally executed, keeping track of all cells undone this way, re-executing the (now modified) cell, and executing all the remembered cells in the order they were first executed. Further research is needed to establish how this relates to live programming [334, 351]. The QL language described in Section 3.6.2 has a live programming environment and forms a natural staring point for this study.

The MiniJava notebook discussed in Section 3.6.1 displays the execution graph of the exploring interpreter, allowing arbitrary roll-backs to explore alternative execution paths. In future work we will explore the ability of the exploring interpreter to support exploratory programming. More generally, we aim to describe algebraic operations over execution graphs for both live and exploratory programming.

The methodology of Section 3.5 starts from a single base language. The methodology is easily generalized to take multiple base languages as a starting point and defining a single sequential language as an extension of all the base languages, which is then used as the basis for a so-called *polyglot* REPL. The definitional interpreter for the sequential extension may not be easy to define, however, when the effects of the phrases of the different base languages are not easily reconciled. In a future study we hope to formulate and demonstrate the more general methodology and to show its benefits to developing polyglot REPLs and notebooks.

RELATED WORK   REPLs have long history and documentation on this history is scattered across sources. The Flexowriter system of Lisp I from 1960 is perhaps the oldest REPL implementation [216]. An early description of REPL behavior can be found in Peter Deutsch's memo on PDP-1 LISP [89]:

Each S-expression typed in will be evaluated and its value printed out.

The PILOT system [341] is one of the earliest and most advanced interactive REPL systems, also based on a LISP, in that it supports fully incremental and interactive

evolution of programs. Teitelman writes that REPL-style interaction with Interlisp happened with the introduction of time-sharing at MIT in 1964 [342]. It is very well possible, however, that earlier Lisps and pre-1968 FORTH implementations [283] had REPL interfaces as well. The earliest programming language REPL that is not a Lisp we could find documentation of is the JOHNNIAC Open-Shop System (JOSS) [316]. Figure 3.5 shows an example of interacting with JOSS.

REPLs have a close relation to computational notebooks, which were pioneered in the Mathematica system [387]. More recently, this style has been adopted in the context of other programming languages. IPython [268] and Jupyter [182] provide a means for computational story telling, where cells containing code are interleaved with output and prose cells. The language workbench framework Bacatá allows a language engineer to provide a notebook feature by reusing existing language artifacts [359]. In Section 3.6.1 we have adapted Bacatá to include the generic exploring interpreter algorithm of which the execution graph is shown in the notebook

Reynolds first employed definitional interpreters as a vehicle for reasoning about languages [289, 290]. His analysis took advantage of the formal similarity between denotational and interpretative semantics [291]. The formal similarity between various approaches to formal semantics is captured by Initial Algebra Semantics [116]. Modular extension mechanisms have been developed for semantics, such as monad transformers [209, 242], entity propagation in Modular Structural Operational Semantics [20], and copy-rules and forwarding in Attribute Grammars[332, 353]. These mechanisms greatly enhance the practicality of definitional interpreters. In modern languages, we see advanced use of monads in Haskell [214, 269], Object Algebras [260] in Java, C# and Scala and intrinsically-typed definitional interpreters in Agda [299].

The usage of an execution graph containing all intermediate configurations of a user's interaction is related to back-in-time debugging [211, 275], also known as omniscient debugging [52, 208], allowing programmers to go back in time of an execution history. In contrast, however, exploring interpreters allow users to go back in *session time*, obtaining both a new run-time state and program state.

CONCLUSION   REPLs provide programmers with a direct interface to a programming language, supporting exploration, testing, and incremental development. All mainstream languages have REPL interfaces, indicating the value they represent to programmers. However, the actual language that is accepted by the REPL is often not well-defined, and engineering REPLs lacks solid design principles.

In this chapter we have surveyed existing REPLs in a feature-oriented domain analysis, showing a wide diversity in feature support. To make the relation between a REPL and its language precise, we have defined and formalized the notion of sequential language, and used it as the basis of a methodology to construct REPL interpreters. The versatility of the approach has been demonstrated in three case studies, one based on MiniJava, and two based on DSLs (QL and eFLINT). The case studies show notebook,

command-line, and client-server REPL interfaces, developed using the methodology by extending base languages and reusing existing interpreters.

The concept of sequential language and its associated language design and engineering guidelines may provide better insight into the essence of REPLs, and promote a principled approach to the construction of REPLs.

MAKING THE INVISIBLE VISIBLE IN COMPUTATIONAL NOTEBOOKS

Computational notebooks are an increasingly popular tool for experts from various fields, not necessarily skilled in software engineering, to experiment with programming and develop software. The kind of interactive and exploratory programming for which computational notebooks are often used is not naturally supported by their design as insights into program state can only be achieved indirectly through executing program fragments and updating one's mental model. In this chapter, we discuss the possibility of defining widgets to improve notebooks by providing direct insights into the program state. The widgets are enabled by previous work in which a novel approach to incremental programming is suggested based on the notion of an exploring interpreter. As examples, we present widgets for visualizing execution history and variable assignments, thereby reducing the cognitive load on end-users, and evaluate the widgets using the cognitive dimensions framework.

4.1 INTRODUCTION

End-user Development (EUD) emerged as a human-computer interaction field in which methods and technologies are studied to enable users to extend or customize their software [210]. EUD has received much attention in recent years due to its focus on empowering people, mostly non-professional programmers, from various domains to create software. There are different ways to support EUD, such as the development of software languages (e.g., high-level programming languages and domain-specific languages), development tools (e.g., IDEs, REPLs, and computational notebooks), development frameworks, Tangible User Interface (TUI), and Graphical User Interface (GUI). There is an enormous potential for EUD since end-users significantly outnumber professional programmers [297].

In this chapter, we focus on computational notebooks, offering end-users a friendly programming environment for simultaneous application and creation of programs. Computational notebooks are cell-based documents that allow end-users to interleave prose, code, and results in a single document. Notebooks have become popular in a wide range of disciplines such as mathematics, physics, data science, programming education, data journalism, and machine learning. They have proven to lower the barrier to entry to programming for novices [303], compared to the traditional toolkits employed by professional programmers such as Integrated Development Environments (IDEs) or plain-text editors and command lines. There are more than 60 notebook platforms [201], with the most popular provided by the Jupyter project [181]. Jupyter notebooks have millions of users, and notebook documents are readily available on

GitHub repositories [304]. Besides their ease of use, notebooks are popular because of their reproducibility, allowing analyses on data to be shared among colleagues and the public [271].

Although notebooks offer manifold features for end-users, little attention has been paid to displaying feedback of the notebook's state, as identified by Chattopadhyay et al. [69]. Likewise, notebooks miss some useful features, such as debugging, offered by traditional IDEs. These features might empower end-users further, when available in a user-friendly manner.

This chapter presents the benefits of using a so-called 'explorer interpreter' as the backend for a computational notebook and explains how to develop interactive *widgets* for a notebook based on an exploring interpreter. As examples, we discuss two widgets – an execution graph and a variable watcher – helping the end-user visualize the program state in order to better predict the effects of subsequent actions. The presented widgets are stepping stones towards computational notebooks with various (generic or domain-specific) widgets directly interacting with the program state.

4.2   INCREMENTAL PROGRAM DEVELOPMENT

Read–Eval–Print Loops (REPLs) are interactive programming environments in which programmers develop programs incrementally by executing code fragments one-by-one, receiving immediate feedback for each fragment. A REPL takes as input the user's code (read), evaluates the code (eval), and displays a summary of the code's effects on the current program state (print). Most REPLs also print the computed value in the case the evaluated code is an expression. REPLs are used for various tasks, such as testing library functions or APIs, debugging programs, learning new language constructs and exploratory programming (discussed below). In Section 3.2, we analyzed 15 REPLs for some of the most popular languages, mapped out their features, and proposed a principled approach to building a REPL for a language as an extension of a definitional interpreter[1] for the language [44]. This work also defines the class of *sequential languages*, in which every sequence of programs is itself a valid program, capturing those languages that naturally support the kind of incremental program development discussed above. This work further concludes that the command-line tool commonly referred to as a REPL is one type of interface for incremental program development that can be built on top of sequential languages, as laid out in Figure 4.1.

Based on a definitional interpreter for a sequential language, an *exploring interpreter* is an algorithm that keeps track of execution context and executes code by applying the definitional interpreter within a chosen context. Execution context (or program state) is represented using *configurations*, in reference to Plotkin's Structural Operational Semantics [273], and is maintained in an *execution graph* as defined in Definition 3.4.1.

---

[1] A definitional interpreter is an interpreter that simultaneously defines and implements the operational semantics of a language.

An edge in the execution graph is between two configurations and is labeled with a program, denoting that the program was executed in the context provided by the source configuration and resulted in the target configuration. The difference between the target and source configuration reflects the *effects* of the program execution represented by the edge. Since the definitional interpreter is for a sequential language, every path in the execution graph is also a valid program (transitivity). The exploring interpreter makes it possible to revisit configurations by changing the execution context to any configuration in the execution graph. The exploring interpreter, therefore, naturally supports exploratory programming.
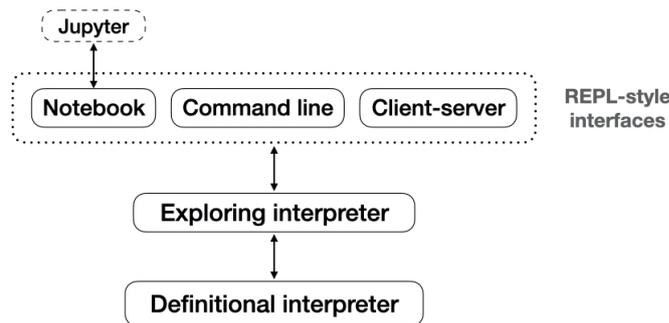
Figure 4.1: Architecture of REPL-style interfaces on top of an exploring interpreter.

REPL-style interfaces are used in a broad variety of contexts, yet they are particularly well-suited for exploratory programming. Exploratory programming is a software development style based on an (open-ended) activity in which the requirements or goals are not fully defined upfront but still to be discovered [284, 345]. In exploratory programming, code is used as a medium for prototyping, and it is during the experimentation process that users find both questions and answers [39]. This programming style is supported by programming environments that allow users to create, edit, and incrementally evaluate partial programs and provide users with feedback during these activities. Designing and implementing a REPL-like interface is cumbersome and might provoke significant engineering costs. The exploring interpreter of [44] provides modularity through indirection, separating language design (to meet the sequential language requirements), implementation (as a 'standard' definitional interpreter) and the maintenance of execution state (by the exploring interpreter). In other words, an exploring interpreter enables engineering REPL-style interfaces with replaceable parts. The hypothesis investigated in this chapter is that a wide variety of interface components (i.e., widgets) can be built on top of the output of an exploring interpreter. As explained earlier, this investigation is done in the context of computational notebooks.

Computational notebooks form a modern incarnation of literate programming in the style of Knuth [183]. These documents consist of a sequence of three types of cells: *documentation*, *code*, and *output* cells [132, 181, 359, 370]. This definition is slightly different from the technical definition of Jupyter, in which the output cell is defined to

be part of the code cell. In notebooks, programs are developed incrementally, executing code cells one-by-one similarly as code snippets are executed in a REPL. The output for each code cell is displayed in a corresponding output cell. A notebook interface can be connected to multiple back-ends to support different languages. Figure 4.2 displays an example of a Jupyter notebook that contains three cells. The first cell contains documentation, the second cell Python code, and the third cell displays the output of the Python interpreter.
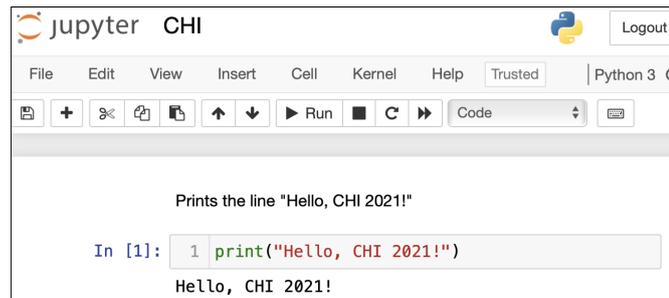
Figure 4.2: Jupyter notebook example.

Notebooks are designed to be written and executed in a linear, top-down fashion. However, notebooks are often used for exploratory programming tasks (e.g., to analyze data), which inherently proceeds non-linearly. When computational notebooks are used for exploratory programming tasks, users wish to modify and re-execute previous cells. However, when these actions occur, users only have access to the information displayed in the notebook (e.g., cell-numbers, input, and output cells); they do not have direct access to the underlying interpreter's state. Thus, users have to keep track of changes and execution order themselves, increasing cognitive load. To support exploratory programming in notebooks, we develop widgets based on an underlying exploring interpreter.

The implementation of the widgets is based on an implementation of a generic exploring interpreter developed in the Rascal language workbench [180]. The notebook interface is implemented using Bacatá [359]. Bacatá is a language-agnostic notebook generator that enables the communication between the Rascal language workbench and the Jupyter notebook platform. A notebook interface in Bacatá is defined using the following Algebraic Data Type (ADT):

```
data Notebook[&T] = notebook(&T initialConfig, &T (str snippet, &T config)
    interpreter);
```

This ADT receives an initial configuration of an arbitrary type &T and a function that takes a code snippet as a string (snippet) and a configuration (config) as parameters and produces an updated configuration of type &T. This definition fits naturally with the exploring interpreter's definition. Each evaluation (interpreter invocation) produces

an updated configuration. In a traditional interpreter, when users execute a snippet, the interpreter returns only the output. Instead, the exploring interpreter produces a configuration reflecting the entire execution context, including output values. From the computational notebook perspective, this approach is interesting because the response to each code evaluation request is the new and previous configuration, reflecting all the effects of executing a code cell. Notebook platforms can use the additional information provided by configurations to help users gain better insights, increase their productivity, reduce errors, and overall, provide a better programming experience.

4.3 WIDGETS IN COMPUTATIONAL NOTEBOOKS

This section presents two widgets, an *execution graph*, and a *variable watcher*. These widgets are developed on top of a notebook interface derived from using an exploring interpreter. Therefore, they have access to all the effects produced after executing a code snippet as specified in the *configuration*. In this context, we use the word widget to refer to an interactive Graphical User Interface (GUI) capable of displaying and manipulating the program state. Widgets used in this section are in a 1-1 relationship with computational notebooks. This relationship transforms the traditional notion of notebooks as a sequence of cells into a document that contains a sequence of cells and input GUI components that reflect the interpreter state. This section uses the Calc language to illustrate the use and benefits of developing widgets on top of an exploring interpreter. Calc is a tiny calculator language that supports basic arithmetic operations. It consists of commands (variable declaration and expression evaluation) and expressions (variables, numbers, addition, and multiplication).

The following code snippet presents the definition of the configuration for the Calc language; it encapsulates the effects that must be stored after executing a valid program. More precisely, this configuration contains the environment (env), the possible output produced by the interpreter (output), and the result of evaluating the current code snippet (val). The environment env stores all the available variables in a dictionary that uses the variables' names as keys and the variables' content as values.

```
data Config = config(Env env, list[int] output, int val);
```

4.3.1 *Execution Graph*

In a notebook environment, when a user executes a cell, the underlying notebook interface (REPL-like interface) returns either the result of evaluating the code snippet or void. Instead, a notebook interface implemented on top of an exploring interpreter, as described in Section 4.2, produces a configuration after each snippet execution. It is important to remark that a configuration encapsulates all the effects of executing a snippet (including both the status of the interpreter and any possible output). In

sum, a configuration reflects everything relevant to the computation as proposed by Plotkin's Structural Operational Semantics [273]. The *execution graph widget* provides a visual representation of the execution graph maintained by the underlying exploring interpreter. In the execution graph, nodes represent configurations resulting from executing code snippets (programs), and the edges represent program executions that make changes to configurations. The widget enables users to 'travel in time' by selecting a previously encountered configuration as the execution context for the code cell executed next.

When a new notebook is created, the *execution graph* only has a single node (root node), representing the initial (empty) configuration, and it has no edges. When users execute a code snippet, the notebook interface receives the snippet and the current configuration. The notebook interface then calls the exploring interpreter to evaluate the snippet and return an updated configuration with all the effects that the snippet produced. Currently, the node labels display the configuration's id in the execution graph. However, these labels should summarize the contents of a configuration so that end-users can use this data to make sense of it without having to depend solely on their mental abilities. To improve the execution graph's readability, its edges are labeled with the corresponding cell number, and the node representing the current configuration is highlighted in green, as shown in Figure 4.3.

Each snippet execution creates an edge in the graph from the current configuration to an updated configuration. If the updated configuration is the same as an existing one, no new node is created, only the edge connecting the nodes, but if the new configuration does not exist in the graph, a node is created. When the number of executions increases, the number of nodes and edges in the execution graph also increases; this impacts the widget's readability, as it can quickly run out of space. One way to mitigate this could be to introduce a zooming in/out capability or define a different visualization mechanism after the graph reaches a certain number of nodes.

The left-hand side of Figure 4.3 shows an example of a Calc notebook, which contains two code cells. The first one assigns the value 1 to a variable x. Hence, the execution graph creates a node and an edge that goes from the root node (initial configuration) to the new node (new configuration obtained after executing the first cell). Then, the second cell assigns the value of the expression x+5 to a variable y. Once more, this creates a new node and an edge; since this is the latest execution, the last node is highlighted in green as the current node (configuration). However, the user has found that the value of y is incorrect, and instead of x+5 should be x+3. To make this change, the user then clicks the 2nd node (the result of executing the first cell) in the execution graph to use that configuration as the current one. After this, the user changes y and executes the second cell again. As a result, the current node has edges pointing to two nodes, the old one and the new one, as shown in the right-hand side of Figure 4.3. If users want to try a different alternative, they can change the current configuration to one of the older configurations, as explained earlier.
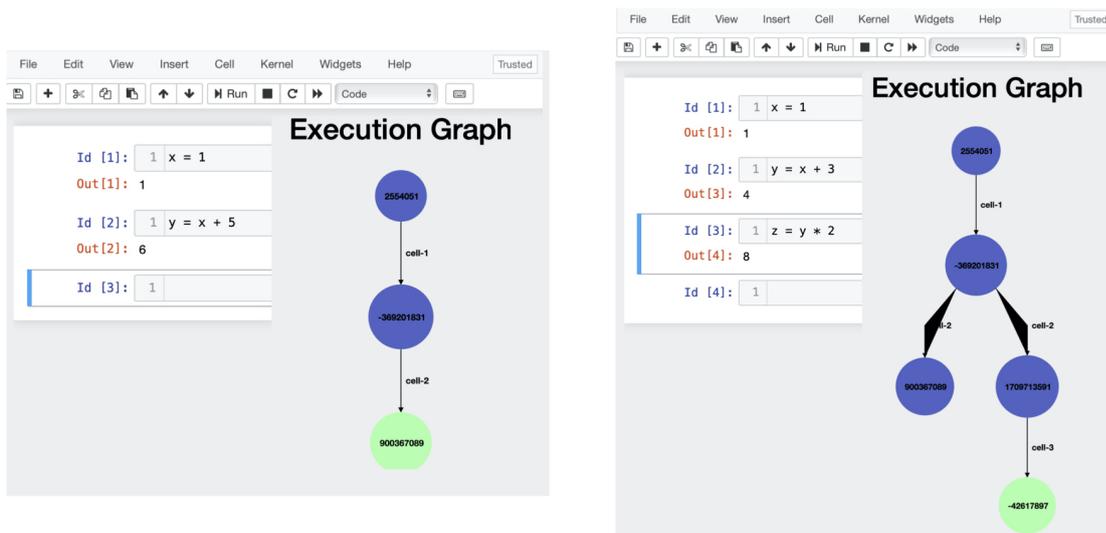
Figure 4.3: Resulting execution graph for the Calc language with a single execution path from the root node (left), and the execution graph using an alternative path obtained from selecting a previous node as the current node (right).

4.3.2 *Variable Watcher*

The execution graph widget discussed in the previous section is a powerful tool that, besides exploratory programming, has the potential to reveal the full execution state and history to the user (when, for example, human-friendly summaries of configurations can be displayed). In this section, we discuss the *variable watcher widget* as an example of a more fine-grained widget providing insights into the current execution state. The variable watcher allows users to read the assignments made to variables, including information about assigned objects (e.g., global variables) and the types or sizes of variables. The current widget can be extended to become an interactive GUI, following prior work, in which users get the flexibility to create/edit programs using both code and GUI [135, 174]. To support this flexibility, coordination between code/GUI is required, making necessary to have a common shared underlying state [174]. For instance, the variable watcher can be extended to become interactive and allows users to use Create, Read, Update, and Delete (CRUD) operations. This provides a different interactive interface for end-users, which is naturally supported by the underlying exploring interpreter, yet this is out of this chapter's scope.

The example of Figure 4.4 shows how the variable watcher for the Calc language displays the environment of the current configuration. The variable watcher is always up-to-date because it is updated by extracting the environment whenever the underlying exploring interpreter changes its execution context. This widget is useful to end-users to

mitigate commons mistakes, such as variable duplication, related to otherwise hidden program states.
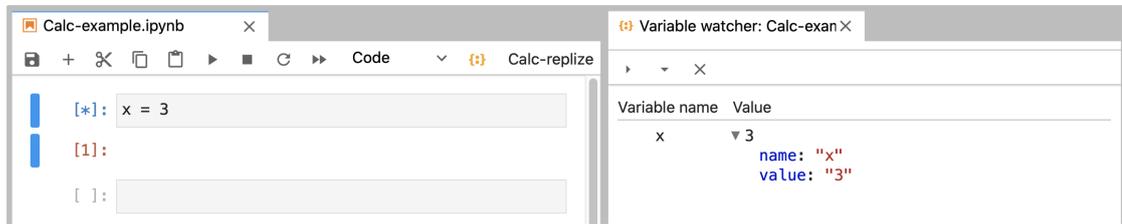
Figure 4.4: Resulting variable watcher for the Calc language.

4.4   EVALUATION: COGNITIVE DIMENSIONS OF NOTEBOOK WIDGETS

We selected seven dimensions from the Cognitive Dimensions Framework [126, 127] for evaluating the impact of adding an execution graph and a variable watcher to a notebook. The selected dimensions are *hidden dependencies*, *premature commitment*, *visibility & juxtaposability*, *secondary notation*, *progressive evaluation*, and *hard mental operations*. Each widget is evaluated using different dimensions because not all are relevant for both widgets, yet some are shared. Table 4.1 presents a summary of the dimensions used for this evaluation.

Dimensions	Description	Widgets	
		Execution graph	Variable watcher
Hidden dependencies	*Essential links between entities are not visible*	✓	
Premature commitment	*Restraints on the order or doing things*	✓	
Visibility & juxtaposability	*Capacity to view components easily, and to place them side by side*	✓	✓
Hard mental operations	*High-demand on cognitive resources*	✓	✓
Role-expressiveness	*The connotation of a component is inferred easily*	✓	
Secondary notation	*Additional information in syntaxes other than the formal syntax*		✓
Progressive evaluation	*Results can be monitored at any moment*		✓

Table 4.1: Summary of the cognitive dimensions used to evaluate the execution graph and the variable watcher. The description of each dimension is based on [126, 127].

The dimensions involved in evaluating the execution graph are *hidden dependencies*, *premature commitment*, *visibility & juxtaposability*, *hard mental operations*, and *role-expressiveness*. Similarly, the dimensions used to evaluate the variable watcher are *visibility & juxtaposability*, *hard mental operations*, *secondary notation*, *progressive evaluation*.

HIDDEN DEPENDENCIES.   Code cells in a notebook are not always executed linearly from top to bottom, and notebooks lack a mechanism for displaying the execution dependencies between cells. The only mechanism offered for this purpose is the input and

output cell ids, yet this is not intuitive enough to understand the notebook's execution flow. The execution graph shown in Section 4.3.1 makes explicit to notebook users the execution dependencies between cells, and may be used to compare alternatives, as shown on the right-hand side of Figure 4.3, which might be handy for exploratory programming tasks. It is important to remark that it only displays dependencies between the executions of cells and not data dependencies.

PREMATURE COMMITMENT.   Notebook users must keep in mind the order in which cells have been executed and their relationships; this forces users to make decisions upfront if they are not sure about the notebook's current status. Thanks to the execution graph, users have access to extra information upfront. The information represented by the graph allows users to understand the current execution state of the notebook. However, the current status of the execution graph can be improved to display additional information that might be helpful for end-users and its look and feel.

VISIBILITY & JUXTAPOSABILITY.   Standard notebooks only display the toolbar and the set of cells in the top and bottom part of the document, respectively. Thus, there are no additional components that might help users develop their programs in a notebook environment. The execution graph is always visible to remind users about the execution flows of the current REPL session. Likewise, as shown in Figure 4.3, the graph is placed side by side, so it does not interfere with the rest of the notebook (documentation, input, and output cells). Similarly, the variable watcher is always visible to end-users, as shown in Figure 4.4. However, it only displays the variable's name and value, yet this can be extended depending on what information is relevant for the users within a domain. It is essential to keep in mind that widgets' juxtaposability depends on the size of the widgets that are being displayed. If both widgets are to be displayed simultaneously, some adjustments must be made because their size depends on the amount of information. The more executions in the notebook, the bigger the graph is; likewise, the variable watcher will require more space as the number of variables increases.

HARD MENTAL OPERATIONS.   As mentioned earlier, in a standard notebook setting, end-users must keep track of the changes made to the notebook's cells, the impact those changes might have in the REPL's state, and the order in which the cells have been executed. The execution graph reduces this cognitive load by making this information available to end-users all the time. Likewise, the main goal of the variable watcher is to unveil hidden information to the user. Therefore, it helps users because they can always observe the environment's current values within a notebook. This information can be used for further executions or to fix previous executions when used together with the execution graph.

ROLE-EXPRESSIVENESS.   The purpose of a notebook interface is evident at first glance; it offers a document based on cells, and after a cell is executed, a result is obtained. However, there is no way to identify the relationship between cells' execution quickly. Therefore, the execution graph is an appealing addition to the traditional notebook interface. Its design makes it easier to understand the execution paths in a notebook session. Likewise, the graph allows users to 'go back in time' and try different alternatives (paths). In a traditional setting, users must restart the notebook kernel; this also means users have to re-execute all the involved cells.

SECONDARY NOTATION.   A variable watcher allows users to query information about the current state of the underlying REPL. However, this GUI can also be adapted in such a way to make changes to the environment. For instance, to change a variable's value directly from such an interface. This does not interfere with the traditional notebook environment but offers an alternative to make this type of data manipulations. In the same way, one can think of other artifacts that enable a secondary notation for concrete actions within a language.

PROGRESSIVE EVALUATION.   Notebooks allow users to obtain feedback after each cell's execution. However, the execution's result is often a single output. With the development of the variable watcher, users can have access to the whole environment all the time. Therefore, the widget keeps up to date with the most recent changes because it is updated after each cell execution to reflect the latest version of the REPL.

4.5   DISCUSSION & CONCLUSION

Computational notebooks have become popular in different communities among different users such as data experts, domain experts, and novice programmers because they have lowered the entry barrier to programming compared to traditional programming environments that require a compile-edit-run loop. However, it is still possible to enrich the end-users' programming experience by revealing the system's state and different ways of creating programs (e.g., using GUIs together with code). For this purpose, we presented in Section 4.3 an example of how software developers can use an exploring interpreter to create notebook widgets that provide additional information (e.g., state of both the notebook and the underlying interpreter) to users. Likewise, the widgets built on top of an exploring interpreter can be extended in such a way that users can interact with them to create programs (based on the configurations maintained by the underlying interpreter). Providing additional information and different interfaces for creating programs within a programming environment is particularly relevant for End-user Development (EUD). This can be achieved by fluid, bidirectional moves between widget actions and executed code [135, 174, 175].

In conclusion, this work is a stepping stone towards improving notebooks by developing widgets capable of providing data about the underlying interpreter's state and creating domain-specific notebook widgets. Both activities are naturally supported by exploring interpreters. To illustrate this, we developed two notebook widgets, as shown in Section 4.3, which are generic, and can be used for various software languages. For instance, in the Geographic Information Systems (GIS) domain, notebooks are used to integrate different distributed components used in the geospatial scientific stack [49], and widgets can be built to provide extra information to users based on information provided by the distributed components. These opportunities are to be explored in future work.

Part III

PROJECTIONAL EDITORS

PROJECTING TEXTUAL LANGUAGES

Projectional editors allow engineers to mix different notations (graphical, tabular, textual, etc.) within the same program. Many existing languages, however, are fully textual and are defined using grammar technology. To allow such languages to be used from within MPS, language engineers have to manually recreate the syntax of a language using MPS concepts. In this chapter we present an approach to automatically convert grammar-based languages to MPS languages, by mapping context-free grammars to MPS concept hierarchies. In addition, parse trees of programs in those languages are mapped to MPS models. As result, MPS users can import textual languages and their programs into MPS without having to write tedious boilerplate code. We have implemented the approach in a tool, Rascal2MPS, which converts grammars in the built-in grammar formalism of Rascal to MPS. Although the tool is specific for the Rascal context, the underlying approach is generic and can be instantiated for other grammar formalisms. We have evaluated Rascal2MPS by generating an importer for a realistic programming language (ECMAScript 5). The results show that useable MPS editors for such languages can obtained, but that further research is needed to improve their layout.

5.1 INTRODUCTION

Language Workbenches (LWBs) [99] are Integrated Development Environments (IDEs) that support engineers in the design and development of software languages [197]. These tools are aimed to improve and increase the adoption of Language-oriented Programming (LOP). LOP is a technique for solving software engineering problems through the use of multiple Domain-Specific Language (DSL) [110]. DSLs are small and simple languages tailored to solve problems in a particular application domain [235]. There are two types of DSLs, internal and external [110]. The first one reuses the concrete syntax of the host language and its parser, much like a stylized library. An external DSL, however, typically requires the implementation of a parser and compiler.

Jetbrains MPS is a projectional language workbench that obviates the need for parsing, and as a result, allows the engineer to define DSLs with a multiplicity of notations, varying from textual, and tabular, to diagrammatic, or prose-like. MPS provides editor support that allows users to directly edit the abstract syntax structures of a language, rather than reconstructing such structure from the linear sequences of characters entered in text editors.

Nevertheless, many existing languages are defined purely textually. For instance, all mainstream programming languages are textual (e.g., Java, C#, JavaScript etc.). But many DSLs, like GNU Make, GraphViz, SQL etc. are strictly textual languages too. To make such existing languages available for (re)use from within MPS, language

engineers have to redefine the syntax of such languages using the concepts and editor features of MPS, which is a tedious and error-prone endeavor.

In this chapter we detail an approach to take an existing context-free grammar (e.g., from a parser generator tool) of a textual language, and convert it automatically to MPS concept definitions. As a result, such languages can be imported into MPS without having to write abstract syntax definitions by hand. Furthermore, the approach supports loading parse trees of existing programs into automatically generated MPS editors, so that they become available for reuse immediately.

Companies in the Eindhoven (The Netherlands) region (e.g., Canon Production Printing, and ASML) have been using DSLs for several years [220]. Some of these companies use textual LWBs, projectional LWBs, or both, such as Canon Production Printing. When companies are using both types of LWBs, it is often desired to reuse existing textual languages within a projectional LWB and vice versa. If such a reuse facility exists, companies will avoid the costs of reimplementing features and maintaining the same functionality in different platforms. Likewise, developers can be more productive from the engineering point of view and invest more time in developing new features or improving existing ones. Finally, the reuse strategy could reduce time to market for new products.

In this chapter, we present an approach towards bridging the gap between textual and projectional LWBs, which has been implemented in the context of the Rascal (textual) and MPS (projectional) language workbenches. Our tool, Rascal2MPS [26], takes a Rascal grammar and converts it to equivalent concept hierarchies and editor definitions in MPS.

The contributions of this chapter can be summarized as follows:

- A generic bridge between textual and projectional LWBs. Employing this bridge, developers can obtain a projectional language in JetBrains MPS from a context-free grammar written in Rascal.

- A mechanism to generate projectional editors from a context-free grammar. This mechanism uses a set of pretty-printing heuristics that takes into account the production rules' structure.

- A tool to import existing programs written in a textual language as projectional models of the generated language.

The structure of this chapter is as follows: in Section 5.2, we describe the motivation that supports this work and the problem statement. Then, in Section 5.3, presents background information about software language engineering. In Section 5.4, we present our solution and its architecture. Then, we evaluate the current approach by comparing an ad-hoc implementation of JavaScript against a generated version (Section 5.5). In Section 5.6, we discuss the limitations of the current approach. We

conclude this chapter with related work and future research directions (Sections 5.7 and 5.8).

5.2 MOTIVATION

A DSL offers programming abstractions that are closer to domain requirements than general programming languages [244]. Likewise, DSLs offer syntax closer to the domain expert's knowledge. DSLs have been around for a few decades, but they have not been widely adopted in the industry yet [88, 192]. The limited adoption of DSLs in the industry is partly due to the lack of mature tools [254, 361].

Nowadays, language engineers have different tools and metalanguages to choose from when implementing a new language. The right selection of such tools is essential for the language's success. Likewise, this means that companies end-up with diverse ecosystems of languages and tools. These tools are continuously changing to support diverse business requirements, depending on what they want to achieve or the organization's needs. Communication between tools and languages is often required to share functionalities among different components. When there is no communication between platforms, developers could reimplement these features. However, reimplementing these functionalities is a cumbersome activity, and it does not fix the problem in the long term because, at some point, it might be required to reimplement those features again.

For instance, there are several textual languages at Canon Production Printing that they have been developing and maintaining over the years. However, they have more recent languages that were developed using a projectional LWB. They have recently found that they require to interoperate languages, which means reusing language concepts across LWBs. This interoperation allows them to address new business needs and reduce the time to market. Therefore, they demand a bridge that supports the reuse and translation of existing languages across heterogeneous LWBs.

5.3 BACKGROUND

In this section, we present some of the basic concepts used in this chapter. The concepts described below are mostly about Software Language Engineering (SLE). Mainly, we focus on discussing the language's syntax and its definition in both textual and projectional LWBs.

5.3.1 *Software Language Engineering*

Software Languages. A software language is a mean of communication between programmers or end-users and machines to develop software. Languages are often divided

into three main components, namely, syntax, semantics, and pragmatics [113, 197]. A language's syntax is a set of rules that define valid language constructs, such as defining a group of rules that captures expressions or statements. The language's syntax can be expressed in a concrete and abstract way. The concrete syntax is designed as the user interface for end-users to read and write programs, whereas the abstract syntax is the interface to the language implementation. The semantics of a language is a mapping between syntactic elements and their meaning. Such mapping can be defined in different manners, such as operational semantics or model-to-model transformations [197]. Language pragmatics describes the purpose of the language constructs, and it is defined informally often in natural language through narrative and examples.

Language-oriented Programming (LOP). LOP is an approach to software development where the main activity in development consists of defining and applying multiple DSLs [93, 110]. Programmers define custom languages to capture aspects of a software system in a structured way. The idea is that each language captures the essential knowledge or aspects of a domain problem so that the productivity increases, and domain knowledge is decoupled from implementation concerns. In other words, a DSL captures the "*what*" of the domain, whereas compilers, code generators, and interpreters define the "*how*".

Language Workbench (LWB). To help language engineers develop software languages, they rely on metaprogramming tools called LWBs. These tools simplify and decrease the development cost of software languages and their tooling [110]. A LWB offers two main features: a specialized set of metalanguages for defining the syntax and semantics of DSLs and affordances to define various IDE services such as syntax highlighting, error marking, and auto-completion. In this chapter, we are going to focus on the former. There are two types of LWBs, namely, textual (also called syntax-directed) and projectional (also called structural) [62, 99, 197]. The main difference between these types is how languages are described and how programs are edited. A textual LWB employs plain text and parsing to map concrete syntax to the internal structures of the LWB. For instance, Rascal uses context-free grammars as formalism [79] for defining the language's syntax. A projectional LWB allows a program's Abstract Syntax Tree (AST) to be edited directly [95]. For instance, MPS uses an AST Concept Hierarchy [62] to define the language's structure, and MPS implements a projectional editor for manipulating programs. A *projectional editor* is a user interface for creating, editing, and manipulating ASTs.

5.3.2 *Syntax of Textual and Projectional Languages*

As mentioned before, a software language's syntax is a set of rules that describe valid programs [197]. Usually, it is divided into two, namely, concrete syntax and abstract syntax. In this subsection, we describe how different LWBs represent both types of syntaxes.

Language	Rascal	MPS
Concrete Syntax	Context-Free Grammar	Projectional Editor Definition
Abstract Syntax	Algebraic Data Type	AST Concept Hierarchy

Table 5.1: Comparison between notations used for describing languages in textual and projectional LWBs.

In textual LWBs, a language's concrete syntax is usually specified using Context-free Grammars (CFGs), while in projectional LWBs, the concrete syntax is expressed as AST projections. Below we explain both approaches and highlight their main differences. To clarify the differences between textual and projectional LWBs, we will use Rascal and MPS. Table 5.1 shows a comparison of the notations used by these two platforms to define language's syntax.

CONTEXT-FREE GRAMMARS   A CFG is a formalism for describing languages using recursive definitions of string categories. A CFG C is a quadruple:

$$C \rightarrow (S, NT, T, P)$$

Where S is the start symbol ($S \in NT$), NT is a set of syntactic categories also known as nonterminals, T is a set of terminal symbols, and P are production rules that transform expressions of the form $V \rightarrow w$. V is a nonterminal ($V \in NT$), and w could be zero or more nonterminal or terminal symbols ($w \in (T \cup NT)$).

For example, a CFG that describes the addition of natural numbers $\mathbb{N}$ is shown below:

$$G = (Exp, \{Exp, Number\}, \{+\} \cup \mathbb{N}, P)$$

The production rules P are defined as follows:

$$start \rightarrow Exp$$

$$Exp \rightarrow Number$$

$$Exp \rightarrow Exp + Exp$$

$$Number \rightarrow i (i \in \mathbb{N})$$

```
start syntax Exp =  number: Nat nat | addition: Exp lhs "+" Exp rhs;

lexical Nat = digits: Natural;
```

Listing 5.1: Concrete syntax of addition and numbers in Rascal.

Listing 5.2: Lexical library.

```
lexical BasicString = [a-z]*[a-z];
lexical Natural = [0-9]+;
lexical String = "\"" ![\"]* "\"";
```

By applying the previous production rules we can write the arithmetic expression $a + b$ (where $a, b \in \mathbb{N}$) as:

$$start \rightarrow Exp$$

$$Exp \rightarrow Exp + Exp$$

$$Exp + Exp \rightarrow a + Exp$$

$$a + Exp \rightarrow a + b$$

$$a + b$$

Once there are no more nonterminals (*NT*), we cannot rewrite the expression $a + b$ because there are no production rules that can be applied. We say that a program is syntactically valid if there is a derivation tree from the start symbol to the string that represents the program.

For instance, the concrete and the abstract syntax of the language described above can be implemented in Rascal, as shown in Listings 5.1 and 5.3, respectively. The first one defines two nonterminals, namely, *Exp* and *Nat*. The *Exp* rule contains two productions, for literal numbers and addition. The *Nat* nonterminal defines natural numbers. The AST Listing 5.3 defines an Algebraic Data Type (ADT) that captures the structure of the language with two constructors: *nat(...)* and *add(...)*. The terminals of the expression grammar (i.e., Nat) are represented using built-in primitive types of Rascal (i.e., int).

Listing 5.3: Abstract syntax of addition and numbers in Rascal.

```
data Exp = addition(Exp lhs, Exp rhs) |  number(int n);
```

```
concept Addition extends    BaseConcept
              implements Expression

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  left  : Expression[1]
  right : Expression[1]

  references:
  << ... >>
```

```
concept Number extends    BaseConcept
              implements Expression

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  value : integer

  children:
  << ... >>

  references:
  << ... >>
```

Figure 5.1: Concept definition of addition (left) and numbers (right)

```
addition {

   left :
      number {
         value : 1

      }
   right :
      number {
         value : 6

      }

}
```

Figure 5.2: Reflective editor for the operation a + b, where $a = 1$ and $b = 6$.

SYNTAX IN PROJECTIONAL LWBS   In a projectional LWB, the syntax is also divided into its concrete and abstract representation. The concrete syntax corresponds to an editor definition, whereas the abstract syntax is defined in a concept hierarchy.

Projectional editors do not share a standard formalism for defining abstract syntax; therefore, each platform provides its own formalism. MPS uses a node concept hierarchy [62]. For instance, the AST representing a language for describing the addition of natural numbers is shown in Figure 5.1. The MPS implementation uses an *Expression* interface and two concepts, namely *Addition*, and *Number*. To represent integer numbers, we use the built-in *integer* data type.

How the users will edit expressions of this kind is defined by an editor definition. However, MPS also offers a generic *reflective editor*, so that every concept in MPS comes with a default editor. A reflective editor is a projectional representation of an AST that developers can use out-of-the-box. An example of an arithmetic expression program using the reflective editor is shown in Figure 5.2.

5.4   APPROACH: PROJECTING TEXTUAL LANGUAGES

This section presents a mechanism for enabling textual languages usage in a projectional editor by generating a projectional language from a grammar. In other words, the current approach translates existing textual languages into equivalent projectional languages, including both structure and editor aspects. Then the translation of existing textual programs into equivalent models of a generated projectional language is discussed. We first show a general overview of the approach. Then, we explain a generic mapping between CFGs and the structure of a projectional language. Afterward, we describe the derivation of a projectional editor from a grammar; we show how to derive the editor aspect for each generated concept in the language structure. Finally, we explain the translation of textual programs to projectional models that conform to a generated projectional language. Although the current approach is implemented using Rascal and MPS, its principles can be adopted in the context of other LWBs.

5.4.1   *Mapping Grammars to Concept Hierarchies*

This section contains the description of the mapping between a grammar and the structure of a projectional editor. The current approach analyzes a CFG, namely, production rules, nonterminal, terminal, and lexical symbols. To illustrate each of the concepts of the mapping, we use the grammar for the *Addition language* shown in Listing 5.1.

Nonterminal symbols. The counterpart of a nonterminal symbol in MPS is an interface.

An interface is a programming concept that may define the public, shared structure of a set of objects (typically described by classes). In MPS, interfaces are represented as concepts and their instances are called nodes. In the same way that interfaces may have multiple implementations (the classes), a nonterminal is "realized" by one or more productions. For instance, in Listing 5.1, there are two nonterminals, namely, Exp and Number. Thus, these two nonterminals map to two interfaces with the same name in the generated projectional language. The definition of the Exp interface in MPS is shown in Listing 5.4.
.

Furthermore, one additional nonterminal that we have not mentioned is the start symbol. Structure concepts in MPS have a property named *instance can be root*. This attribute indicates whether the concept can be used to create an AST root node [62]. In our mapping, we take the start symbol of the grammar and create a concept in MPS. This concept will have the property *instance can be root* set to true. For instance, in Listing 5.5, we show an example using the expression language, assuming we have a start symbol Program with a single production, prog.

Productions. A nonterminal rule has one or more productions. As we mentioned before, a nonterminal in a CFG is mapped to an interface concept in MPS. Therefore,

Listing 5.4: Definition of the Exp interface in MPS.

```
interface concept Exp extends <none>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>
```

Listing 5.5: Mapping a CFG start symbol into a MPS concept.

```
concept prog extends <default> implements Program

  instance can be root: true
  alias: <no alias>
  short description:  Exp

  children:
  expression : Exp[1]
```

to keep the relationship between a nonterminal and their productions, we map each production as an MPS *concept*. Each *concept* must implement the interface of the nonterminal. Moreover, the AST symbols in the production rule are mapped to either the *children* or the *properties* field. When the symbol is a nonterminal, it is defined in the *children* field, and when the symbol is terminal or a lexical, it is mapped in the *properties* field. Note that symbols that are only relevant to concrete syntax, such as keywords and operator symbols are not mapped here, since they are not part of the abstract syntax; they will be used to define the editor aspects (see below).

For instance, addition (Listing 5.1) is a production rule of the nonterminal Exp. This production rule is mapped into an MPS concept that implements the Exp interface. The resulting concept in MPS is shown in Listing 5.6.

Lexicals Lexical define the terminals of a language and are typically defined by regular expressions. Rascal allows full context-free lexicals, but here we assume that all lexicals fall in the category of regular languages that can be defined by regular expressions.

To ease the mapping between Rascal lexicals and MPS concepts, we define a Rascal module that contains a set of default lexicals. These lexicals define the syntax of identifiers, string literals, and integer numbers. Developers can use these lexicals in

Listing 5.6: Result of mapping a production rule to a concept in MPS.

```
concept addition extends <default> implements Exp

  instance can be root: false
  alias: +
  short description:  Exp + Exp

  children:
  lhs : Exp[1]
  rhs : Exp[1]
```

Listing 5.7: Lexical mapping.

```
concept digits extends <default> implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  nat: Natural

  children:
  << ... >>

  references:
  << ... >>
```

their Rascal grammars, but it is also possible for users to include their lexicals. In this case, developers must describe the mapping to MPS manually.

Each lexical is mapped to a concept, like any other nonterminal, and a *constrained data type*. To illustrate this, Listing 5.1 contains Nat's definition, which consists of a single production, called digits. This production rule references Natural, which is one of the predefined lexicals (Listing 5.2). As a result, the lexical Nat is translated into a concept, called digits (Listing 5.7), and a *constrained data type*, called Natural (Listing 5.8). The digits concept has a single property of type Natural, a *constrained data type* capable of capturing natural numbers using the regular expressions engine of MPS.

Listing 5.8: Result of mapping a Rascal lexical to an MPS constrained data type.

```
constrained string datatype: Natural

    matching regexp: [0-9]+
```

Listing 5.9: Concept mapping for a list of symbols.

```
concept groupExp extends <default> implements Exp

  instance can be root: true
  alias: <no alias>
  short description:  Exp

  properties:
  << ... >>

  children:
  exps : Exp[0..n]

  references:
  << ... >>
```

List of symbols. In CFG, it is possible to define a group of symbols of the same type, often expressed using Kleene's star (∗) and plus (+). Kleene's operators (star and plus) are unary operators for concatenating several symbols of the same type. The first one denotes zero or more elements, and the second one denotes one or more elements in the list. The current approach detects both operators (Kleene's star and plus) in productions. The operators are represented in MPS as children of a concept with cardinality zero-to-many ($0..*$) and one-to-many ($1..*$), respectively. For instance, let us add to the language shown in Listing 5.1 the following production:

```
start syntax Exp = … | groupExp: Exp∗ exps;
```

This production defines zero or more expressions (`Exp`). The resulting mapping of the production `groupExp` is shown in Listing 5.9.

5.4.2 *Mapping Grammars to Editor Aspects*

This section presents the mapping between a grammar and the editor aspect in MPS. For creating the editor aspect of the language, we use the language's layout symbols, namely, literal and reference symbols. In this context, a reference symbol is a pointer to a nonterminal symbol (which can be lexical or context-free). come with the language.

Literals. Literal symbols may be part of productions to improve the readability of code or disambiguate. They form an essential aspect of the concrete syntax and can be leveraged to obtain projectional editors.

To create an editor, we first take each production rule; we look at each symbol and keep track of its order. It is essential to keep track of the order because it affects how the editor displays the elements. In this process, we consider two types of symbols, namely, *literals* and *references*. If the symbol is a literal, it is added to the *node cell layout*

Listing 5.10: Generated editor for addition.

```
<default> editor for concept addition
  node cell layout:
    [- % lhs /empty cell: % + % rhs /empty cell:  % -]

  inspected cell layout:
    <choose cell model>
```

Listing 5.11: Editor mapping for a list of symbols.

```
<default> editor for concept groupExp
  node cell layout:
    [-
      (- % exps % /empty cell:   -)
    -]

  inspected cell layout:
    <choose cell model>
```

as a placeholder text. Moreover, this is used to define the syntax highlighting of the resulting editor. The literals are displayed with a different color to show the users that they are reserved words of the language. As a result, the current approach offers a binary coloring scheme: keywords are blue and the remaining symbols in black. Instead, if it is a nonterminal symbol, we create a *reference*.

For example, the production rule that defines the addition between natural numbers has three symbols: *lhs*, +, and *rhs*. Following the approach, we first take the *lhs* symbol and create a reference to its type Exp; then, we take the literal, +, and copy it to the editor, and finally, we create a reference to the *rhs* symbol, which is also of type Exp. Listing 5.10 shows the generated editor for addition. This editor has two references, namely, *lhs* and *rhs*. Editors use references to access concept properties. For instance, in the editor, the reference lhs creates a link to the lhs children in the addition concept. Moreover, the editor, for *addition*, has a literal (+) in between the two references. The literal is shown as a placeholder text for users to write expressions like 5 + 6.

List of symbols. The editor aspect for a list of symbols (zero-to-many and one-to-many) is based on creating a collection of cells. More concretely, each list of symbols is translated into an *indent cell* collection. Listing 5.11 shows the generated editor aspect for the groupExp production.

5.4.3 *Editor Improvement – AST Pruning*

Having defined a mapping from CFGs to the editor aspect in projectional languages, we will improve the generated projectional editor. The editor can be improved by pruning the grammar to enhance IDE services (e.g., auto-completion). To prune the grammar, we eliminate chain rules (also known as unary rules) from the productions. To eliminate the chain rules, we first collect all the productions with a single parent and which are referenced once in the grammar. Then, we merge the single reference with its parent.

To illustrate this process, let's consider the following production:

$$A \rightarrow A|b|c|d$$

Long production rules are often split into smaller production rules for readability. For example, a language engineer can also write the previous production as:

$$A \rightarrow A|B$$

$$B \rightarrow b|c|d$$

The second alternative impacts the language's structure because it introduces a new nonterminal B. This new nonterminal is translated in the AST as an extra node. To illustrate the difference between both versions, Figure 5.3 shows a tree view of the ASTs. From the right-most AST in Figure 5.3, we observe that node B is referenced once in the language. Thus, production $A \rightarrow B$ represents a chain rule. This chain rule is translated to the end-users as an extra keystroke to access the leaf nodes b, c, d via B. If we remove the chain rule, we avoid creating an extra node (B) before accessing the terminals (b, c, d) in the projectional editor.

For example, if users want to create a node b, they can call auto-complete, and they will obtain two options, A or B. Based on the AST shown in Figure 5.3, they select to create a node B. However, they have not reached b yet. Thus, they must press tab-completion again, and then they get all the options of B: b, c, and d. In contrast, if we prune the chain rule, meaning we remove concept B, we can omit the second tab-completion because all the options will be visible from the first tab-completion. Removing chain rules from a grammar impact both the structure and the editor of a projectional language since removing a concept means the editor of such concept is no longer needed. As a result, we enhance the user's interaction with the projectional editor by removing the chain rules.

```
        A                       A
      / | \ \                   / \
     A  b c  d               A     B
                                  / | \
                                 b  c  d
```

Figure 5.3: Tree-based view comparison.

5.4.4 *Translating Textual Programs into Projectional Models*

We extend the approach to translating existing textual programs into projectional models. This extension's motivation is that we want to offer a mechanism for importing existing textual programs into the generated projectional language. We did not consider a manual translation because it is cumbersome, and tools can automate it.

To this aim, we applied the same approach proposed for generating languages. However, instead of only using a grammar as input, it takes both the program and the grammar. We use the grammar for creating a parser; then, the parser creates a parse tree of the program. Both Rascal and MPS offers support to write and read XML files, so we define an XML schema to serialize and deserialize parse trees as XML files. The former acts as an intermediate representation that supports the communication between platforms. The current approach is implemented in Rascal and MPS. However, it is possible to support other platforms by implementing the XML schema.In the textual world, the schema serializes the parse tree; while in the projectional world, the projectional LWB deserializes the XML and uses it to create the projectional model.

The current approach uses the XML file as the input of an MPS plugin. The plugin traverses the XML tree and creates a model that conforms with the generated language. If the translation is correct, the generated model should be a valid instance of the generated projectional language.

5.4.5 *Architecture*

The approach to bridge textual and projectional LWBs contains five components: Rascal2XML, XML2MPS, *XMLImporter*, *ImportLanguage*, and *ImportProgram*. The solution has been implemented using Rascal MPL and Jetbrains MPS. We consider two different architectures for the implementation of the current approach. The first one was based on integrating Rascal directly into MPS, including Rascal as a Java library in MPS. This architecture allows us to call Rascal parsers directly from MPS. However, this approach does not allow reusability, and this integration should be repeated for any textual LWB. Instead, the second architecture uses an intermediate format to communicate

between a textual LWB and MPS. In the following paragraphs, we describe each of the components of this architecture and how they interact with each other. All the code is available on a GitHub repository[1].

Rascal2XML. This module is written in Rascal, and it is responsible for generating an XML representation of Rascal grammars and existing textual programs. This module produces an XML file that is used as input for the module XML2MPS.

XML2MPS. This MPS project holds the logic for generating MPS language definitions and model instances. It is responsible for creating MPS concepts and interfaces from an XML file. Both `ImportLanguage` and `ImportProgram` use this library.

ImportLanguage. is an MPS plugin that enables the import of languages. It creates the Graphical User Interface (GUI) for importing a textual language. The GUI displays a pop-up that takes the grammar (in XML format) as input, calls the *XMLImporter*, and produces a projectional language.

ImportProgram. is an MPS plugin that enables the import of programs. This plugin takes as input an XML file that contains a program, and it produces a projectional model. To create the projectional model, this plugin relies on the XML importer to read the XMLFile and in XML2MPS to create the MPS nodes.

XMLImporter is a Java library for traversing the tree-like content of the XML files. This is used to map textual languages to projectional languages and translate textual programs as projectional models.

5.5 CASE STUDY

In this section, we present a case study to evaluate our approach. The language we have chosen for this purpose is JavaScript (ECMAScript 5) because there is an existing implementation of it for MPS, and it allows a proper validation of Rascal2MPS. First, we explain the definition of the language. Then, we show how we create a mapping between the textual language and the generated projectional language. Afterwards, we generate a projectional editor based on the language's concrete syntax. Finally, we import existing textual programs as valid MPS models that conform to the generated projectional language. This section concludes with a brief discussion based on results.

---

1 https://github.com/cwi-swat/rascal-mps

5.5.1 *Language Description*

So far, we have presented a way of applying the approach to a toy language of expressions. Now we will apply it to a well-known and widely used language. To show the applicability of the approach to a real-world language, we reused the existing grammar definition for JavaScript, included in Rascal's standard library. This grammar can be found in GitHub[2]. This evaluation aims to use a Rascal implementation of the JavaScript grammar, and obtain the equivalent language in MPS.

First, we must sanitize the existing grammar to meet our solution's constraints, as described in Section 5.6. It is essential to mention that this sanitization process is entirely manual. In this grammar, the sanitization process consists of adding labels to all the production rules and variable names to all symbols; and changing lexicals to use either one of our predefined lexical types or a user-defined construct. The resulting sanitized grammar can be found on GitHub[3].

We then used this grammar as input to generate the XML that encodes the grammar definition into the intermediate format. This XML representation is also available on GitHub [4]. The XML file can then be imported into MPS. In MPS, we use the plugin that we built, and we use the XML file as an input to successfully generate the projectional version of JavaScript.

To evaluate our generated version of JavaScript, we decided to compare it against an ad-hoc MPS implementation of such a language called *EcmaScript4MPS* [5]. *EcmaScript4MPS* is a fine-tuned implementation of JavaScript for MPS. In other words, the implementation considers how developers use JavaScript editors and the features offered for JavaScript in IDEs. For comparing both implementations, we show several examples of language elements and programs of both implementations. For the rest of this section, we will refer to the generated version as JsFromRascal and the MPS ad-hoc implementation as JsManual.

5.5.2 *Editor Aspect*

To compare the editor of both languages, we present how a program looks like in both editors. The JsFromRascal program was created using the approach described in Section 5.4.4. This approach takes a textual program as input, the tool parses it and produces an XML file with the resulting parse tree. It is important to mention that we did not tweak the resulting program; we used the generated version as-is. Figure 5.4 shows the resulting program using the JsFromRascal editor.

---

2 https://bit.ly/3JDiP70

3 https://bit.ly/32JtRHp

4 https://bit.ly/32UZKfQ

5 https://github.com/mar9000/ecmascript4mps

In contrast, the program for JsManual was written by hand because we did not have a mechanism, like the one described before, for arbitrary textual programs. However, the hand-written program is the same as the one used for JsFromRascal. The resulting program in the JsManual editor is shown in Figure 5.5

```
function                              for
  substrings                          ( var
                                        i = 0
(                                     ;
  str1                                  i < slent
)                                     ;
{ var                                   i ++
    array1 = []                       )
  ;                                       { temp = "" ;
  for                                     for
  ( var                                   ( var
    x = 0                                   j = 0
    y = 1                                 ;
  ;                                         j < array1 . length
    x < str1 . length                   ;
  ;                                         j ++
    x ++                                )
    y ++                                  { if
  )                                         ( ( i & Math . pow(2,j)
    { array1 [ x ]                            )
      = str1 . substring ( x              )
        y                                     { temp += array1 [ j ]
      )                                         ;
      ;                                       }
    }                                       }
  var                                     if
    combi = []                            ( temp !== ""
  ;                                       )
  var                                       { combi . push ( temp
    temp = ""                                 )
  ;                                           ;
  var                                       }
    slent = Math . pow ( 2              }
      array1 . length                 console . log ( combi . join("\n")
    )                                 )
  ;                                   ;
                                    }
```

Figure 5.4: The substring JavaScript program displayed using the JsFromRascal editor.

As can be seen from Figures 5.4 and 5.5, the program in the JsFromRascal editor takes up more lines of code than its counterpart in JsManual. According to the JavaScript standards, the JsManual editor makes the program look more readable due to the ad-hoc implementation of the editor, which places break lines and whitespaces in the right place. The JsFromRascal editor splits up statements and expressions into several

```
program substring
--------------------------------------------
function substring(str1) {
  var array1 = [],
      x = 0,
      y = 1;
  for (; x < str1.length; x++) {
    array1[x] = str1.substring(x, y);
  }
  var combi = [];
  var temp = '';
  var slent = 'Math.pow(2, array1.length)';
  var i = 0;
  for (; i < slent; i++) {
    var temp = '',
        j = 0;
    for (; j < array1.length; j++) {
      if (i & 'Math.pow(2,j)')
        {
          temp += array1[j];
        }
    }
    if (temp !== '')
      {
        combi.push(temp);
      }
  }
  'console.log(combi.join("\n"))';

}
```

Figure 5.5: The substring JavaScript program displayed using the JsManual editor.

lines based on the implemented heuristics. Instead, the JsManual editor does not break these language constructs into several lines. However, it forces users to define variables outside *for* statements due to the language's name resolution implementation.

Another difference between the editors is the usage of the dot operator (.). This operator is often used in programming languages to access fields or methods. For instance, JsFromRascal identifies it as a binary operator (e.g., '+', '-'), and therefore the editor introduces whitespaces before and after the dot operator. This is an example of the limitations introduced by the heuristics; they are rigid. A customization mechanism might be needed to make such heuristics more flexible; thus, they can be adapted to different languages and scenarios.

In sum, the JsManual editor is more appealing, and visually, it looks more like a textual program written using a plain text editor than the one generated using JsFromRascal. This kind of difference was expected because the JsManual editor is

implemented in an ad-hoc way to offer the best experience for this language, while the JsFromRascal editor is obtained through a generic tool that works for various languages. However, the JsFromRascal editor can be manually fine-tuned to achieve the desired editing experience. The knowledge of the JsFromRascal editor depends entirely on two core elements, the information contained in the grammar and the set of heuristics applied to such grammar. On the one hand, the creation of ad-hoc editors from scratch, such as the one made for JsManual, is a cumbersome activity. On the other hand, a generated editor speeds up editors' development process because they use generic abstractions that can be applied to several languages, so that developers can focus on fine-tuning the generated editors on edge cases based on platform-specific features and the language's coding styles.

5.5.3   *Program's Usability*

Now we are going to discuss the usability aspects of both editors. Here we only focus on the ease of creating and editing programs with the editors mentioned above. First, we investigate the tab-completion menu, which is one of the critical aspects of a projectional editor since it allows users to navigate through the language's structure (AST). In Figure 5.6, we present a code completion menu for a *for* statement in JsFromRascal, and in Figure 5.7, we present the equivalent using JsManual. Both editors show similar information: the concept's name and a brief description. However, the JsFromRascal editor also displays the structure of the child nodes of such a concept, which might help developers understand how to use concepts or remember the concept's syntax.
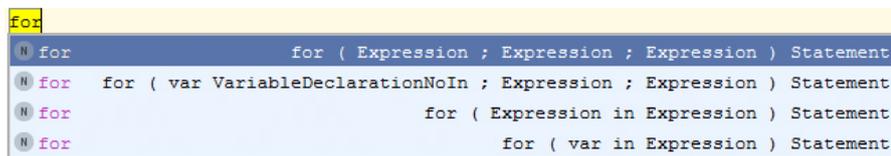

Figure 5.6: JsFromRascal editor tab-completion menu of a *for* loop.
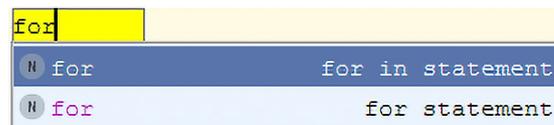
Figure 5.7: JsManual editor tab-completion menu of a *for* loop.

5.5.4 *Discussion*

PROJECTING GRAMMARS AS LANGUAGE STRUCTURES.   The first goal and building block for this project is to recreate the structure of a language in two different LWBs. This goal was previously achieved and explained by Ingrid [369]. We wanted to try a different solution in which we do not directly integrate both platforms, but instead, we define an intermediate format to make the solution more general. Section 5.4.1 describes the process for mapping a textual language definition into a projectional language definition. As shown in Section 5.5.1, the current approach works, yet some considerations must be taken into account to generate a proper language. We understand that the way we treat lexicals might be cumbersome since the complex structure's mapping must be manually defined. We also think this could be solved by defining some pre-processing strategies to capture lexicals and generate them into the second platform.

EDITOR ASPECT – LANGUAGE USABILITY   The editor aspect of a language is essential because it is the user interface to the language. Nevertheless, implementing a good editor is cumbersome. As shown in Section 5.5.2, usability is one of the main differences between ad-hoc and generated implementations. In the generated version, we applied heuristics from the literature (e.g., well-known formatting and pretty-printing approaches) to try to identify production rule patterns generically. However, these heuristics have limited power, and of course, they might not fit every language, especially if we compare them against custom implementations. Nonetheless, with the current approach, we show that it is possible to apply existing heuristics to create projectional editors based solely on the language's grammar. Besides, the current approach considers the language's structure to generate a projectional editor that, in some cases, might be more appealing than the reflective MPS editor.

To improve the current approach, we could have implemented more heuristics or define a mechanism for customizing them. We might also require additional information other than the information contained in the grammar. Also, languages' coding style and user feedback are fundamental to improve the quality of generated editors. In other words, we need more information to implement the heuristics in a less rigid fashion, and therefore improve the editor generation.

5.6 LIMITATIONS

This section discusses the limitations of the approach, the rationale behind them, and possible solutions to overcome them. These limitations are based on assumptions and constraints in the grammar. Besides, there is also a technical limitation related to how the mapping is implemented.

> **Summary - Grammar Preconditions**
> - Non-terminal symbols name and production rules labels within a grammar must be unique.
> - Symbol labels within a production rule must be unique.
> - Lexicals can be either one of the MPS predefined data types or the lexical must be defined by hand using the lexical library.
> - Each production rule and each symbol within a production rule must be labeled.

1. The names of the non-terminal symbols in a grammar must be unique. In other words, the current approach does not support the definition of two concepts with the same name. The rationale behind this is that the name of a non-terminal symbol is used to define an interface concept in the generated MPS language, and the production labels are used to create concepts. One way to avoid this constraint could be defining a renaming scheme that can detect and fix name conflicts. However, this solution might introduce a side effect on the language's usability; projectional editors use these names for IDE services such as tab-completion, so they must be descriptive enough for end-users. Also, other language components must be refactored according to the renaming mechanism. Therefore, we did not implement an automatic renaming scheme, and we preferred to include it as a limitation of the current approach.

2. In the mapping between a Rascal grammar and an MPS language, symbol labels are used as variable names, either for `children` or `references` in MPS concepts. These names should be unique within the same concept, yet not for the whole language. For instance, if we define concepts *A* and *B*, both can contain a reference of a child named *name*; however, *A* cannot have more than one child or reference called *name*. In other words, symbol labels can be reused across concepts but not within the same concept.

3. Lexicals are a challenging concept to deal with because there is no standard way of defining them. However, it is possible to make some assumptions on regularity and define a set of constraints to translate lexical between platforms in an automatic way, but this requires considerable effort. As a result, we did not want to restrict regular expressions, so we included lexicals that represent MPS built-in types (e.g., string, int) to the lexical library. The current approach does not limit users from defining custom lexicals. However, users must manually define a mapping between the custom lexical defined in Rascal and the right translation for MPS. Section 5.4.1 describes the details on how to support custom-defined lexicals.

4. It is required to label all the production rules and symbols within a production rule because the approach uses the labels for naming concepts or children reference

fields. A solution could be to generate placeholder names, yet this introduces other issues such as non-descriptive names and name matching issues when importing existing textual programs.

5. The current approach does not take advantage of name resolution, especially for code completion, which is a keystone for projectional LWBs. For instance, in MPS, concept hierarchies do not rely on trees' definition; instead, they use graphs.

6. The current implementation supports the mapping of lists and separated lists of symbols into MPS language concepts (editor and structure aspects). However, the mapping for separated lists is partially implemented. The current approach treats separated lists just as a list. As a result, the separator symbol is ignored for the generation of the editor.

The current approach does not support language nor program evolution. In other words, the current approach considers languages as standalone units. It does not consider that changes might happen to the language. For example, if a developer uses a textual language *A* and generates a projectional language *A** inside MPS, the current approach only accepts valid programs according to *A*. If there are changes to the original language *A*, those changes cannot be patched in the generated versions. This forces to re-generate the whole language from scratch or make changes by hand. Some changes do not break the importing of programs:

- Addition of language constructs to the grammar and then using them in a program. This means that the plugin for importing programs, *ImportProgram* (Section 5.4.5), will not find such elements. As a result, the plugin notifies the user.

- Modification of existing language constructs (e.g., adding or removing parameters). As expected, this type of change often ends up in a failure.

In sum, language engineers and users, in general, should be aware of the language's version and the version used to define programs. We see this problem as an opportunity for future extensions of the current approach to supporting languages and programs' evolution.

5.7 RELATED WORK

Projectional LWBs allow users to manipulate the programs' AST directly; therefore, parsing technology is no longer needed. In contrast, textual LWBs parsing is essential. This section presents state of the art in grammar to model transformation and editor generation.

5.7.1 *Grammar to Model*

The generation of models from grammars is essential for the current approach. Thus, we identified the following related work in this direction.

Ingrid [369] is a project that attempts to bridge the gap between textual and projectional LWBs. Their approach uses ANTLRv4 [263] as textual LWB and JetBrains MPS as projectional editor. Ingrid is implemented as a hybrid solution in Java/MPS project. Ingrid bridges textual and projectional LWBs in three steps: first, the grammar must be parsed, and relevant information about the structure and other required language elements is stored as linked Java objects. Secondly, the stored structure is traversed, and equivalent MPS model nodes and interfaces are constructed. Finally, an editor is generated for each MPS Language Concept Node. There are some high-level similarities between Ingrid and Rascal2MPS. Both projects perform the steps taken for parsing, gathering information about the language, generating an intermediate structure to represent the language, and finally generating a model from the said intermediate structure. The main differences are in the architecture, design, and implementation choices of both projects, which have various consequences for using the respective tools. The main architectural difference is in the choice of the intermediate structure. Whereas we chose an external file-based format (see Section 5.4.5), Ingrid uses an internal representation of linked Java objects. This decision enables them to use the ANTLRv4 parser implemented in Java and the ability of MPS to call into Java executables directly. Thus, the Ingrid MPS plugin can call the parser and start the data extraction process internally. In contrast, Rascal2MPS keeps both LWBs separate; they can communicate only through an external intermediate format. Some of the advantages of not using an intermediate format are:

- The solution becomes a one-step process, making it more efficient for the language engineer.

- All implementation is done on one side of the bridge (projectional LWB), simplifying the development.

- The language engineer does not need to maintain both the textual and projectional LWB.

However, this approach has a significant downside: the projectional LWB must call the grammar parser directly. Thus, there is a strong coupling between the projectional LWB and the specific grammar parser. In the case of Ingrid, the MPS plugin calls into the Java ANTLR parser. However, the ANTLR parser is not the only one. If we wished to extend Ingrid to support Rascal, we would need to replace ANTLR parser calls with Rascal parser calls. This can lead to several problems:(i) The architecture must allow this replacement. This can be partially solved using interfaces and abstractions over the parser, but the problem of potentially different APIs remains. A complete mapping from

ANTLR parser function calls to Rascal parser function calls would have to be made in the worst case. (ii) The parser needs to be implemented in Java. ANTLRv4 already has a Java-based parser and is a prime candidate for integration with the Java-based MPS. However, this is not necessarily true for any given textual LWB. If one is not available, the language engineer would either have to implement the parser in Java, or find some way to expose the parsing features to a Java environment.

Rascal2MPS addresses the problem of bridging the gap between the textual and projectional worlds in a generic-fashion. In other words, neither side of the solution is aware of the other; they communicate only through the intermediate file-based format. This format serves as a contract between the different parts of the solution. If the intermediary file is generated from an ANTLR-, Rascal- or Xtext-based grammar is irrelevant to the implementation on the side of the projectional LWB.

Another difference between Ingrid and Rascal2MPS lies in the editor generation. While Ingrid does identify the problem of usability of the reflective editor and discusses several solutions, such as heuristics or prompts during the import process, they have not been implemented. Ingrid only generates an editor containing the node's structural elements, i.e., the literals and references to other nodes. It is then left up to the language engineer to apply whitespace to the editor manually. Rascal2MPS goes further and applies heuristics to apply whitespace during the import process automatically. While this does not eliminate the need to edit the editor definitions manually (Section 5.5), it can save time given the right set of heuristics. Finally, Ingrid does not address the problem of language artifacts, i.e., programs created within the textual world. Thus, even after a language has been imported, programs are written using said language in the textual LWB that needs to be manually recreated as MPS models of the imported language. Rascal2MPS does implement the ability to construct MPS program models using textual source code.

Wimmer et al. [386] describe a generic semi-automatic approach for bridging the technological space between the Extended Backus–Naur Form (EBNF), a popular grammar formalism, and Meta-Object Facility (MOF), a standard for model driven-engineering. In this approach, an attribute grammar describes the EBNF structure and the mapping between EBNF and MOF. Then, it is used to generate a Grammar Parser (GP). This GP can then be used to generate MOF meta-models from grammars. However, this approach fixates on MOF as the target meta-model directly. In the case of going between LWBs in separate worlds, we do not want to be specific in the target. Instead, Rascal2MPS uses an intermediate format and makes the source and target formalism up to the implementation. Another downside of the given approach is that it requires grammar annotations and additional manual improvements of the generated model to refine the generated model. We seek to limit the actions of the language engineer, especially concerning the source grammar. The Gra2Mol [147] is another project which seeks to bridge the gap between the textual grammar and model worlds. The authors define a domain-specific model transformation language that can be

applied to a program that conforms to a grammar and generates a model that conforms to a target meta-model. This language can be used to write a transformation definition consisting of transformation rules. In this way, the presented approach abstracts over the generated meta-model, which would be quite useful in our use-case, as we would be able to give the meta-model of the target LWB as input with the transformation definition. In practice, however, this runs into problems when the desired target model is specific rather than generic. For example, the standard storage format for JetBrains MPS is a custom XML format. The models contain much information tied specifically to MPS, such as node IDs and layout structures. Generating these from outside of MPS would be quite tedious and would introduce a dependency on the MPS model format, which may change. Thus, it is best to interact with the MPS model from within MPS itself, where MPS can do the heavy lifting of generating the models.

5.7.2 *Editor Generation*

Editor generation is an essential step in bridging the gap between textual and projectional LWBs. It is closely related to the well-known pretty printing problem in the grammar world. Grammar Cells [367] is an extension of MPS that offers a declarative specification for defining textual notations and interactions in a projectional editor. Implementing editors with this extension makes it easier to offer a text-like editing experience; thus, it is widely adopted by the MPS community. Our current implementation does not use Grammar cells because we restricted our approach on a plain MPS installation. However, this extension's adoption is part of the roadmap for the next iteration of the current implementation.

Van de Vanter et al. [354] identifies part of the core problem between the textual and model-based approach. From a system's perspective, a model-based editor allows for easier tool integration and additional functionality. However, language users are often more familiar and comfortable with text-based editing. In this chapter, the authors propose a compromise based on lexical tokens and fuzzy parsing. This is not unlike what is offered by MPS. MPS Editors are highly customizable and can be made to resemble the text-based editing experience closely.

As introduced by van den Brand et al. [55], the BOX language for formatting text is closely related, as the heuristics for generation white space between language elements is reused in this project. The BOX language is further used in other work on pretty-printing generic programming languages, such as GPP (Generic Pretty Printer) [86], which constructs tree structures of a language element's layout that can be used by an arbitrary consumer.

Syntax-directed pretty-printing [302] also identifies several structures for creating language-independent pretty-printers. In this approach, a grammar extended with special pretty-printer commands is used as input to generate a pretty-printer for such a language. The generated pretty printer can then be reused for any program

written in the language the pretty printer was generated for. The annotated grammar approach does limit the form the final pretty-printer can have due to the lack of options. Also, annotating an entire grammar can be tedious work. We attempt to limit the required user interaction with the source grammar in our approach, although we did not eliminate it.

Following this research line, Terrence et al. [264] propose *Codebuff*, which is a tool for the automatic derivation of code formatters. *Codebuff* is a generic formatter that uses machine learning algorithms to extract formatting rules from a corpus. This is a neat approach because, as we mentioned before, source code formatting is subjective, it depends on each programmer's style, and it changes across languages. For example, in Section 5.5.2, we showed that applying the same heuristics for any language does not always produce a good editor. Therefore, we consider tools like *Codebuff* as inspiration for future work. We could benefit from their techniques and knowledge to generate editors in a flexible and highly configurable way and perhaps learn from existing source code examples.

5.8   CONCLUSIONS AND FUTURE WORK

In this chapter, we presented an approach to bridge the gap between textual and projectional LWB. We defined a mapping between textual grammars and projectional meta models; this mapping (Section 5.4) produces the structure and editor aspects of a projectional language. Moreover, our approach allows users to reuse textual programs by means of translating them to equivalent MPS models (Section 5.4.4). To validate our solution, we used as a case study a Rascal grammar of JavaScript (Section 5.5). Based on the grammar definition, we generated a projectional version of JavaScript. To verify the correct mapping of the generated language, we successfully imported existing valid textual JavaScript programs into MPS. In Section 5.6, we discuss some of the limitations of the current approach.

Language evolution is a crucial aspect to look at in the future. Since the current approach assumes that the generation is done only once, we ignore the fact that the textual language and the projectional generated version might change. Then we consider that keeping track of these changes and transferring/applying these changes to the other is essential If there are changes in the grammar after the projectional language generation, developers must regenerate the whole language, which may lead to losing information (if changes were made on the generated language).

Similarly, this applies to programs written in such languages. We consider that a mechanism for maintaining both versions is worth investigating as future work to keep a bidirectional mapping. Language engineers can switch from one platform to another without losing information. Our approach offers support for a unidirectional mapping from textual to projectional. We believe that a bidirectional communication is required. Because depending on the language, one may benefit more from having a textual or

a projectional version of the language. Therefore, to support both sides' changes, we require a bridge to create a textual language from a projectional language. Moreover, to complete the circle, a way of keeping track and propagating changes in both worlds will be required. To avoid losing or reimplementing existing features.

As we described in Section 5.5.4, the usability of generated editors is one of the critical aspects that should be addressed in future research. We found that we can generate editors with limited capabilities (that do not consider domain knowledge or existing formatters). Therefore, we consider as future work, to explore artificial intelligence techniques (e.g., machine learning or programming by example) to improve the existing editor (in the style of [264]), maybe by identifying patterns in existing programs or commonalities in the grammar's structure to guide or to customize the generation of the editor aspect.

Part IV

BLOCK-BASED EDITORS

WHAT YOU ALWAYS WANTED TO KNOW BUT COULD NOT FIND ABOUT BLOCK-BASED EDITORS

Block-based environments are visual programming environments, which are becoming more and more popular because of their ease of use. The ease of use comes thanks to their intuitive graphical representation and structural metaphors (jigsaw-like puzzles) to display valid combinations of language constructs to the users. Part of the current popularity of block-based environments is thanks to Scratch. As a result they are often associated with tools for children or young learners. However, it is unclear how these types of programming environments are developed and used in general. So we conducted a systematic literature review on block-based environments by studying 152 papers published between 2014 and 2020, and a non-systematic tool review of 32 block-based environments. In particular, we provide a helpful inventory of block-based editors for end-users on different topics and domains. Likewise, we focused on identifying the main components of block-based environments, how they are engineered, and how they are used. This survey should be equally helpful for language engineering researchers and language engineers alike.

6.1 INTRODUCTION

In the past decade, end-user programming environments have become more popular and more relevant. End-users significantly outnumber professional programmers [297]. These environments allow end-users to create and adapt software, enabling them to achieve a myriad of tasks. Without them, most of these tasks would not be possible unless one has a background in computer science or software engineering. Block-based environments are part of this set of end-user visual programming environments. One of their main characteristics is the editor, which presents the language constructs to the end-user as graphical elements that resemble Lego blocks. Each of these blocks is characterized by visual signifiers (e.g., color and shape) that hint end-users of its semantics and its (possible) connections to other blocks. An example of a block-based representation of an `if` statement is shown in Figure 6.1. The benefit of having a block-based editor is to offer a programming experience based on What-You-See-Is-What-You-Get (WYSIWYG) and the impossibility of syntactic errors [246, 276, 378, 382]. Moreover, these editors support different block-based programming paradigms, such as *configuration*, *serial*, *parallel*, and *event-driven* [122].

The popularity of block-based editors have increased in recent years, partially due to Scratch's popularity (23rd most popular programming language [61]). However, languages that provide such a type of editors are not new, yet block-based editors have been mainly used and associated with computer science education or applications for
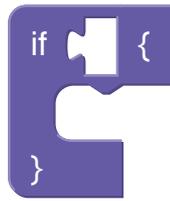
Figure 6.1: Block-based representation of an `if` statement.

children. This chapter explores whether this is true or not. In case this claim is not true, we explore how these programming environments have been adopted beyond the realms of education or children. Moreover, this chapter identifies block-based environments' main components to understand them and increase their adoption in different domains and for different user groups; and studies whether the development of block-based environments is supported by specialized language engineering tooling (e.g., language workbenches).

To have a clear overview of the landscape of block-based environments and understand how they are developed, we conducted a systematic and a less-systematic literature review. A Systematic Literature Review (SLR) collects and summarizes all the existing research evidence of a domain and identifies possible gaps in current research [177]. Initially we started with the less-systematic method, in which we sought block-based environments and their features. We ran into the limits of this ad-hoc method and continued with an SLR to identify possible gaps in current research [177].

Since there exists no primary conference or journal focused on block-based environments, we expect that papers on this topic are spread over different academic communities with different characteristics. The papers we found in the venues will frame the answers to the research questions about block-based environments.

The contributions of this chapter are summarized as follows:

- A systematic literature review on block-based environments   which provides an overview of the main features of block-based environments, the landscape in which these programming environments are used, publication venues, programming languages used in their development, and the most popular environments (Section 6.3).

- A deeper (qualitative) understanding of block-based environments and their components (Section 6.3.2).

- An understanding of how block-based environments are implemented and the tools and languages involved in their development (Sections 6.3.3 to 6.3.5).

- A non-systematic tool review of block-based environments that presents some of the most relevant features of these programming environments (Section 6.4).

We describe the threats to the validity of this study in Section 6.5 and a discussion of the results obtained in this study in Section 6.6. The non-systematic literature review

fills some gaps of the systematic literature review and vice versa. We concludes with a discussion of related work, conclusions and future research directions (Sections 6.7 and 6.8).

6.2 SYSTEMATIC REVIEW: MOTIVATION AND METHODOLOGY

A literature review about block-based environments can be approached from different perspectives. Different perspectives are relevant because block-based environments are used across different domains and user groups, and developed by engineers with different backgrounds as well. Our agenda is to make block-based editors part of the default set of services offered by language workbenches [110] and to investigate how editor generation technology (metaprogramming) is used to support this. Therefore, our primary motivation in this literature review is to understand in which contexts are block-based environments used and how exactly they are developed.

We aim to make block-based environments available for all Domain-Specific Languages (DSLs) developed using a Language Workbench (LWB). LWBs have shown benefits by offering specialized (generative) technology for developing software languages and their tooling. LWBs are used to define both the syntax and semantics of languages. Based on these two, language engineers can also semi-automatically derive a full-featured Integrated Development Environment (IDE) for their language [53, 56, 67]. We believe that block-based editors can also be derived from existing language components. To enable that research we need a clear set of requirements to frame the contributions of a block-based LWB and we must understand how block-based environments are best implemented. The current SLR helps in achieving these goals.

6.2.1 *Need for a Systematic Review*

Coronado et al. [74] published a literature review about using visual programming environments for programming robots. In their work, they present an overview of the environments that use a visual editor for programming robots; this considers all sorts of graphical elements to represent language constructs, including block-based editors. However, their main target was to understand the tools used to program robots. A similar approach can be used for other purposes, for instance, to identify the programming environments used for teaching computational skills. Our view is that it is essential to understand block-based environments in general; this includes understanding their features, applications, and technologies involved in their development.

We aim to increase the adoption of the block-based metaphor beyond the realm of computing education, make its development part of the set of generic services offered by specialized tooling for creating software languages (i.e., language workbenches), and explore further applications in industrial settings.

6.2.2   *Protocol*

This section presents the protocol defined for the systematic review. We followed Kitchenham et al. [177, 178] guidelines for conducting a systematic literature review in software engineering. The primary goal is to summarize existing evidence in literature of the development and usage of block-based environments, in different disciplines. The secondary goal is to identify possible gaps between the development of these visual programming environments and existing tools and technologies specialized in developing software languages and their tooling.

6.2.2.1   *Scope*

Visual programming environments are environments that rely on graphical editors for building programs. Different notations are adopted by these environments, such as flow charts, UML diagrams, and block-based editors. In this survey, we focus our attention on the latter notation.

Block-based environments are not new, but the popularity of Scratch [240] has inspired a new wave of visual programming environments. These environments are characterized by representing language constructs with graphical blocks that resemble Lego blocks. Moreover, these environments offer visual cues that help users understand what are the possibilities for connecting blocks.

6.2.2.2   *Research Questions*

The research questions addressed in this study are:

> **RQ 5.0** *What are the characteristics of the papers that present block-based editors?*
>
> **RQ 5.1** *What are the components of a block-based environment?*
>
> **RQ 5.2** *What are the tools used to develop block-based environments?*
>
> **RQ 5.3** *How are block-based environments developed?*
>
> **RQ 5.4** *What languages offer a block-based editor and what are these languages used for?*

The motivation for the meta question **RQ 5.0** is that we expect publications on block-based editors to be scattered acros many different (types of) venues: from fundamental computer science all the way to applications in other academic domains such as medicine, and everything in between. The answer to **RQ 5.0** helps to frame the answers to the following research questions. Research questions **RQ 5.0**, **RQ 5.2**, and **RQ 5.3** are answered through the systematic review. **RQ 5.1** and **RQ 5.4** are answered using both the systematic and the non-systematic approach.

6.2.3 *Search Process*

Languages that use a block-based editor are becoming popular outside the academic world for their ease of use. For instance, commercial robots, programmable micro-controllers, and applications for children use them as an effective end-user interface. Consequently, many of these languages have been developed outside the academic world, which means that there are language implementations that do not have a corresponding academic publication. Vice versa there exist academic publications about languages which do not have an implementation (anymore).

Therefore, to obtain a complete overview of the landscape, it is essential to include both academic and non-academic tools in this literature review. Therefore, we decided to follow a combined search process that is both fully systematic and less-systematic. For the fully systematic process, the first author systematically searched for peer-reviewed papers in computer science academic databases. The less-systematic process was conducted using standard Google search queries. In some cases, some tools reference other tools, so we also used this information. Following this approach, we found 30 different relevant block-based environments.

We consider using Google scholar for the systematic approach, but unfortunately, it provided more than 2.6k results, which is more than what we can deal with. Therefore, we reduced the search space to the four primary academic databases in computer science and software engineering, namely, IEEE, ACM, Elsevier, and Springerlink. The selected academic databases are shown in Table 6.1. They were selected because these databases are well known, and they have proceedings of the leading journals and conferences on which block-based environments have been applied, such as education, software engineering, human-computer interaction, and end-user programming.

6.2.4 *Queries*

To identify and understand languages that offer block-based editors, we used the following *search string* in the academic databases:

```
block-based language OR blocks-based language OR block-based languages OR
blocks-based languages OR block-based programming OR blocks-based programming
```

We used the search string mentioned above for all four academic databases. A summary of the number of results obtained from each database is presented in Table 6.1. Table 6.2 presents a summary of the type and number of publications obtained across all the databases. The publication type *Other* aggregates different types of publications such as demonstrations, posters, magazine columns, tutorials, outlines, living reference work entries, panels, conference description, editorials, and non-peer-reviewed technical

reports. Details about the inclusion or exclusion criteria for the relevant proceedings are explained below in Section 6.2.5.

Source	# Results
IEEE Xplore ®Digital Library	128
The ACM Digital Library	272
Elsevier ScienceDirect ®	55
Springerlink ®	213
Total	668

Table 6.1: Number of publications obtained per academic database.

Publication type	# Papers
Conference paper	369
Journal	143
Other	50
Chapter	44
Abstract	40
Short paper	22

Table 6.2: Number of publications per type.

6.2.5 *Inclusion and Exclusion Criteria*

This section presents the criteria we used for both the systematic and the less-systematic approach.

NON-ACADEMIC   We included solely tools that can be used at the moment of the systematic review, (i) Open-source tools. (ii) Commercial tools with free trial. This includes languages and tools that can be accessed only by contacting the authors, as described on the tool's website.

ACADEMIC   We reviewed the title and abstract of each paper manually to remove all papers that certainly were not featuring languages with block-based environments. The proceedings used in this literature review are all peer-reviewed articles related to block-based programming in the broad sense, published between January 1st, 2005 and August 1st, 2020. Note that we are interested in all articles related to block-based interfaces, so we included all articles that used or mentioned block-based languages or block-based programming even if they present applications or studies of the block-based metaphor solely.

We excluded articles on the following topics: (i) Visual languages that do not feature a block-based editor (ii) Studies not written in English (iii) Frame-based editing [186] unless they provide a connection to block-based editors (iv) Data-flow programming (v) Form-filling programming (vi) Wizardry metaphor [101] (vii) Duplicate articles that present the same tool without adding a fresh perspective.

Finally, we excluded reference work entries, living reference work entries, and educational papers unless they introduce a new tool, a language, or an extension to an existing tool or language that uses a block-based editor.
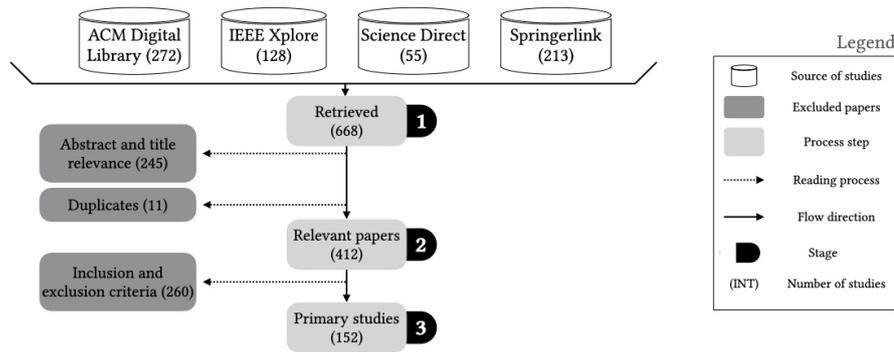
Figure 6.2: Selection procedure for the systematic literature review.

6.2.6 *Selection*

To identify the relevant publications to be included as part of this SLR, we performed three-step filtering (Figure 6.2) on the results obtained from all the databases using the string query mentioned before. We took each result in the first filtering phase and we evaluated its relevance based on the title and the abstract only. Only papers that include something about block-based environments where kept. For each excluded paper, we wrote a motivation about why it was discarded. After this process, the number of papers was reduced from 668 to 423. After removing 11 accidental duplicates we ended up with 412 papers. The second filter step starts with all the papers that resulted from the first filter. In this phase we defined nine yes/no questions based on our research questions. The nine questions are shown in Appendix B.1. Then we counted the number of yes answers for each individual paper. Based on this count we chose a threshold to include a paper for the subsequent filtering step.

Since answering the nine questions is a manual task, the second author double-checked a random selection of ten papers by following the exact same protocol. We measured the degree of agreement between both authors and we calculated *Cohen's kappa* coefficient [71]. This statistic is used to measure the degree of accuracy and reliability in statistical classification. Both authors agreed to include five papers and exclude four papers. However, the first author decided to include one paper that the second did not. To quantify this: there was 90% agreement between the authors and Cohen's kappa was 0.8. According to the guidelines proposed by Landis and Koch [199], a 0.8 Cohen's coefficient means that there is a *substantial agreement* between the parties.

This literature review's primary focus is to provide a landscape of languages and tools related to block-based environments. Therefore, the main criteria to include a paper is to introduce a language or a tool that uses a block-based environment. If that is the case, the paper is included even if the number of yes answers is not greater than the threshold. If the paper does not include a language or a tool, we use the accumulated result to determine whether the paper is included. Thus, a paper that

does not introduce a tool or a language must have more than four positive answers. As we did in the first filtering phase, we always record why a paper is discarded.

The second filter's resulting papers are then the ones on which the current survey is based; this means those are the papers from which we extract the data for further processing and discussion. As a result of the second filtering phase, we excluded 260 papers from the 412 we had after the first filtering. As a result we analyzed 152 papers in the data extraction phase. During data extraction, we retrieved different elements, such as the type of publication, details about the block-based environment (e.g., elements of the editor and its position on the screen), and all kinds of editor implementation details. All the data was collected in a spreadsheet and its content was then analyzed and processed by different means using scripts that aggregate the raw data. The result of this process is shown and explained in the following sections.

6.3   SYSTEMATIC REVIEW OF BLOCK-BASED ENVIRONMENTS

In this section we answer the research questions (Section 6.2) using the data collected from the 152 papers on block-based editors.

6.3.1   *RQ 5.0: What are the characteristics of the papers that present block-based editors?*

This section presents demographics of the papers included in this survey. Particularly, we present the venues in which the included papers were published, the number of papers included per year, and the number of papers per country. Table 6.3 presents a summary of the venues that contributed the most number of papers. For readability we present only the categories that contain venues that contributed at least two papers. The complete list of categories and venues is listed in Table B.2.

To get a quick overview of the most important venues we ordered them in Table 6.3 by ranking them by "popularity". Moreover, we manually classified them into 18 categories. For the classification process we tried two semi-automated alternatives using a more systematic approach, namely (a) calculating the document distance between calls-for-papers of each venues and (b) using Google's Cloud Natural Language API [1] to classify each call-for-papers. The bottom-line is that both approaches did not produce accurate results and so we went back to the manual classification. We report on these negative results nevertheless, as they might be useful to others researchers that are working on an SLR.

We extracted the text in the call for papers of a random sample of venues to use these to test the two automated approaches. In this step, we notice that not all venues present a clear list of topics (e.g. the ACM CHI conference). For the first approach, we calculated the document distance between two calls for papers from the same field.

---

1 https://cloud.google.com/natural-language

Category	Venue	# Papers
Human computer interaction	Conference on Human Factors in Computing Systems (CHI)	13
	Conference on Interaction Design and Children	11
	International Conference on Human-Computer Interaction (HCII)	4
	International Journal of Child-Computer Interaction	3
Programming / Human computer Interaction	Blocks and Beyond Workshop (Blocks and Beyond)	13
	Symposium on Visual Languages and Human-Centric Computing (VL/HCC)	9
	Journal of Visual Languages & Computing	2
	International Symposium on End User Development (IS-EUD)	2
Education	Technical Symposium on Computer Science Education (SIGSE)	10
	Global Engineering Education Conference (EDUCON)	5
	Computational Thinking Education	2
	Education and Information Technologies	2
	Workshop in Primary and Secondary Computing Education	2
	International Conference on International Computing Education Research	2
	Conference on International Computing Education Research	2
Distributed computing	International Conference on Computing, Communication and Networking Technologies	5
Robotics / Education	International Conference on Robotics and Education (RiE)	4
Accessibility	Conference on Computers and Accessibility (ASSETS)	2
Programming	Science of Computer Programming	2
Security	International Conference on Information Systems Security and Privacy (ICISSP)	2
Software engineering	International Working Conference on Source Code Analysis and Manipulation (SCAM)	2

Table 6.3: Summary of venues that contributed at least two papers to the survey.

By manually verifying documents which were either far apart, or close, with our own understanding we noticed nothing but noisy results. Apparently the variety of topics in calls for papers goes far beyond the variety of topics of what a conference is about.

To explore this further, we removed all the other text from the call from papers, and we calculated the document distances based only on the research topics mentioned in the call for papers. However, this did not improve the results, and the document distance between two venues from the same field was not too close (false negatives). And, in many cases even, comparing venues from distinct fields produced closer distances (false positives).

The second approach used the same input data. We used the default Google's classification categories on the same texts, and the results were indeed accurate (correct), but they were not precise enough (vague). I.e. most of the venues were classified as "computer science".

After these failed attempts to automate and objectify our classification, we continued with a manual classification process. Table 6.3 shows that the venues that contributed the highest amount of papers are CHI and 'Blocks and beyond', with 13 papers each. The former is a venue about human factors in computing systems, including interaction, visualization, and human-computer interaction topics. Thus, it is a clear connection between these topics and the benefits offered by block-based environments. The latter venue is exclusively focused on the development and use of block-based environments. Therefore it is a perfect match for the study we present in this survey.

The papers included in this study are from different domains such as Human-Computer Interaction (HCI), Education, Design, Software Engineering, Robotics, and Security. Based on all the venues that contributed at least one paper, we expect our paper collection process to be rather complete for this study since we have publications from a variety of heterogeneous sources and topics. Likewise, this study includes different types of proceedings as shown in Table 6.2.

To understand the papers' demography, we computed the number of papers that we included in our study per year, as shown in Table 6.4. This figure shows that the number of papers per year has increased, having its peak in 2019. It is important to remark that the current survey's search process solely included papers published before August 1st, 2020. This probably explains why the number of papers in 2020 is lower than in 2019. With Table 6.4, we can observe an increase in popularity on topics related to block-based environments.

Moreover, we computed the number of papers published per country. To compute this information we used the nationality of the first author as presented in the paper, and then we calculated the number of occurrences per country. We can observe that the United States is the country that contributed the highest number of papers, followed by the United Kingdom with 64 and 14 papers, respectively. It is essential to mention that the gap between the number of papers contributed by the US is more than four times the number of UK papers. It is also interesting to observe that we have some degree of
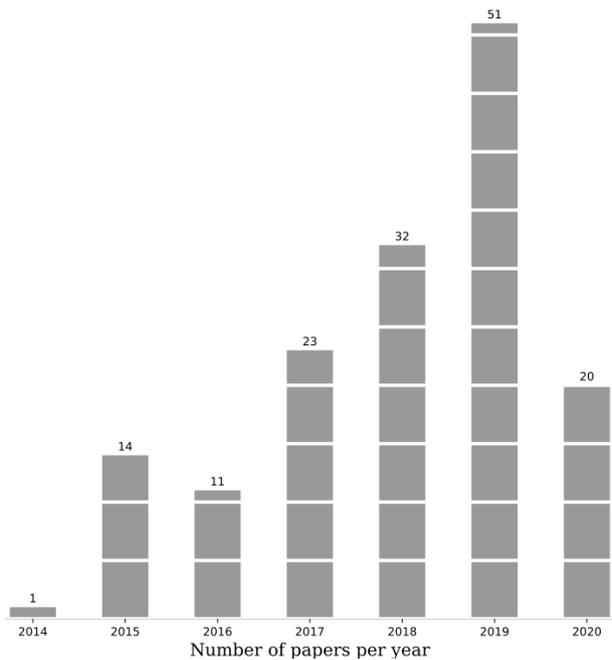
Table 6.4: Summary of the number of papers published per year.

Publication Type	# Papers
Study	31
Languages	95
Extension	27

Table 6.5: Summary of the number of papers included in this study per type of proceeding

diversity in the authors' nationality; there are authors from different continents —North America, South America, Europe, and Asia. Antarctica, Africa, and Australia are not represented. The complete list of papers per country is presented in Appendix B.4.

While analyzing the papers, we decided to tag them using three categories *study*, *language*, and *extension*. We defined these categories to classify the papers based on their content. The first category, *study*, is used to group papers that study aspects of using or implementing block-based environments and do not present a new language or tool that uses the block-based metaphor. The *languages* category is used to group all the papers that present a new language that includes a block-based editor or tools that support the development of block-based environments. Finally, the *extension* category groups papers that do not introduce a language but introduce new features to existing block-based editors. Table 6.5 presents a summary of the number of papers per category.

Based on the previous information, the reader can observe that the included papers come from a wide range of topics, types of publications, and authors from different parts of the world. In the next section, we will present in more detail findings and information that we obtained by analyzing and gathering data from the corpus of papers, and that helps us answer the research questions defined in Section 6.2.2.2.

Summary RQ 5.0

- Publication of block-based environments is spread among different communities, however they are most present in education, human computer interaction, and programming venues.
- The number of publications that present block-based editors have been increasing since 2014. This is supported by the importance of programming in the last years among different people, including students and non-professional programmers.
- Authors from many countries publish papers that use block-based environment. However, the country that contributes the most number of papers to this study is the United States, followed by the UK.
- In this survey, we classified the 152 papers based on their goal in three main categories, studies, languages, or extensions. Most of the papers included in this study are papers that introduce a language (95), followed by studies of the usage of block-based editors (31) and, finally, papers that introduce extensions to existing block-based environments (27).

6.3.2   *RQ 5.1: What are the components of a block-based environment?*

This section addresses research question **RQ 5.1** based on the data collected. For this purpose, we used the papers' classification from the previous section and we took the ones from the *languages* group. From the total number of papers we considered a subset of 95 papers (Table 6.5).

Based on the different features offered from all the block-based environments in this study, we developed a feature diagram [164] that summarizes the most common features found across different platforms. The complete set of features of block-based environments is shown in Figures 6.3 and 6.4. To ease the diagram's readability, we split the *editor* feature into a separate diagram, as shown in Figure 6.4. Figure 6.3 shows the first part of the diagram. Here the reader can observe features related to the functioning of the platform. For instance, *code execution mode* and the *type* of block-based environment. Then, Figure 6.4 presents details of the block-based editor.

In the feature model, we used two types of features, mandatory and optional. The first is used for standard features (depicted as a box in Figures 6.3 and 6.4), and the latter for unique features (depicted as a box with a blank circle on top). The root node in Figure 6.3 represents a block-based environment, and each of the leaf nodes in the feature diagram displays the number of block-based environments that support that feature and the percentage of tools that support it among all the papers. For instance, *Computer (76, 80%)* means that 76 block-based environments are deployed for computers, which is 80% of the papers used for creating this diagram. All the block-based environment's children nodes are described below.

TYPE.   There are mainly two types of block-based environments, *tools*, and *languages*. The former refers to utilities that help the development of such environments. Instead, the latter are languages that come with a block-based editor.
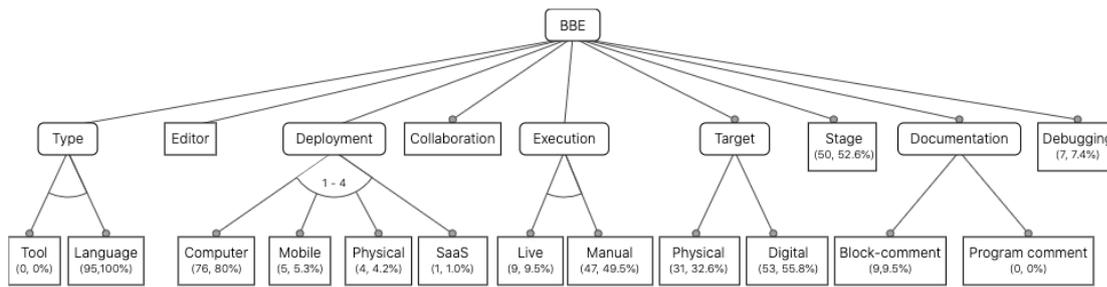
Figure 6.3: This is the top-level of the feature diagram.
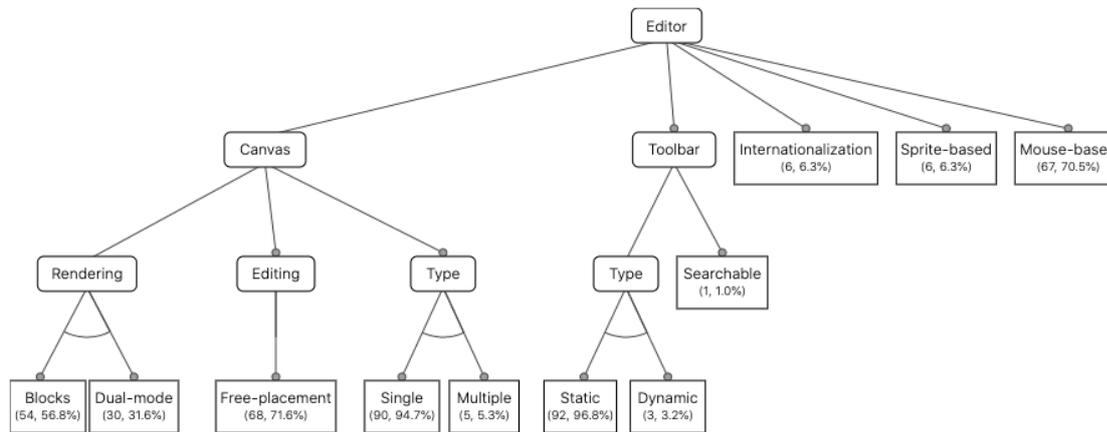
Figure 6.4: This feature diagram shows a zoom-in into the editor aspect of a BBE.

EDITOR. Block-based environments provide a block-based editor, but, we identified that some tools also support a hybrid editor (text and blocks), which means that it is possible to interact with the underlying language either through a blocks editor or text-based editor. Based on this, 69 of the studied tools support a block editor only, while 15 support both blocks and text editor [9, 10, 21, 24, 36, 38, 48, 75, 90, 137, 169, 176, 202, 329, 368]. The remaining ten tools do not mention it at all.

DEPLOYMENT. A block-based environment can be used through a heterogeneous set of devices (e.g., laptops, tablets, and wearables). Therefore, we investigated what device do block-based environment users write or develop their programs with. The majority of tools are used through a browser-enabled PC (76), five through mobile devices (e.g., smartphones), four by manipulating physical elements, and one as Software as a Service (SaaS). In nine of the tools, it was not clear which type of device the users have to develop their programs.

COLLABORATIVE. This feature represents whether a block-based environment supports mechanisms for users to collaborate in the development of programs. From

the studied papers, 90 tools do not offer such capabilities. Instead, the remaining five tools do support this feature.

EXECUTION. There are different ways of executing programs. This is not different in a block-based environment. Based on this study, we identified mainly two execution modes, *manual* and *live*. After finishing the development of their programs, a manual execution means that users have to press a button to launch the execution of the program by the underlying language processors. Instead, a *live* execution mode does not require a manual intervention by the user to execute programs. The platform is capable of live executing the programs as users develop them. From the tools, 47 use a manual execution mechanism, nine use a live execution, and the remaining 39 tools do not mention which execution mode do they use.

TARGET. As mentioned before, block-based environments are used in different settings. This includes the real and the digital world. Thus, sometimes the effects of running a block-based program are displayed on the screen, but sometimes they are shown via hardware, and sometimes both. We investigated this fact and found that 53 tools present some form of results in a digital way, 31 using hardware (physical), and three using both. For the remaining eight tools, it is not clear from the papers which one do they use.

STAGE. Block-based editors are used in different environments. In some environments, the effects of running a program are represented in the real world (e.g., hardware effects). However, other contexts in which the results are displayed as software (e.g., animation). This feature characterizes if the block-based environment has a dedicated pane for showing the effects of running a program. Of the 95 tools, 50 of them have a dedicated pane for rendering program results; the remaining 45 do not have such a pane. Since there is no standard way or location for placing a stage, we investigated the most common place in which block-based environment developers place this component. The preferred location for the canvas is the right-most side of the screen, with 22 tools. Then, 19 tools place it to the left-most part of the screen. Two tools place it in the center; similarly, two tools have it in the bottom part. Finally, only one tool has the stage on the top part of the screen.

However, the following four tools do have a stage, but the paper is not clear wherein the screen it is located. Below, we present the details for these four tools. Catrobat [251] uses a different layout because it is a mobile app. Behavioral Blockly [13] does not show the whole block-based environment. Some images show the programs, and others that show the stage, but not the whole workspace. In VEDILS [249] there are different screens for showing the stage and editing the program. For the mBlock [205] tool it is not clear where the stage is located from the paper's screenshots.

DOCUMENTATION. Documentation is an essential aspect of software. In traditional text-based programming environments, it is possible to add comments anywhere

in the program as long as it does not introduce syntactic errors. In a block-based environment, this is more restricted due to the projectional nature and the visual components. Therefore, we identified mainly two types of documentation. One is used to add documentation to specific blocks (*block-comments*), and the other documents a complete program/script (*program comments*).

BLOCK-COMMENTS As introduced before, block-comments are the comments added to specific blocks. This could be either a group of blocks stacked up together or a single block. From the studied papers, nine tools allow users to add comments per block, while the other 86 tools did not mention it explicitly.

PROGRAM COMMENTS. This feature is presented to show whether the tools allow users to add comments to complete scripts/programs. We found that none of the tools found in this study support adding comments to block-based programs.

DEBUGGING. Traditional software development tools support the debugging of programs. This is no different for block-based environments; however, we found that not all block-based environments support debugging features. From all the tools, only seven tools come with debugging features. The remaining 87 do not mention it; we assume they do not offer such capabilities.

Next, we present in detail the features that are part of the block-based editor Figure 6.4.

CANVAS. The canvas is where users create their programs; it is where they drop the blocks that constitute programs. All block-based environments offer a canvas for building the programs. Most of the papers (70) show their canvases, but some (24) papers did not present screenshots that show the canvas explicitly. Some of these publications that did not present the canvas present block-based programs. The canvas location indicates wherein the screen is this component situated. For 49 of the tools, the canvas is located in the center of the window; 16 have it on the right-most part; one in the left-most part of the screen, and one have it in the bottom part. As explained before, the remaining 27 tools do not mention or display their position.

CANVAS TYPE. Some environments provide more than a single canvas for creating programs. Therefore, we look at the papers, and we found that five tools do use multiple canvases, and the remaining 90 either only offer a single canvas or do not explicitly mention/show support for multiple canvases.

RENDERING. This feature means that the block-based environment displays programs using only blocks or dual-mode (text and blocks). 54 of the block-based environments display programs using only a block-based representation, 30 tools support a dual-mode, and the remaining 11 tools do not mention anything about it.

EDITING. A canvas allows users to build programs by placing blocks on it. However, this does not mean that all block-based editors use a 2D space. From all the tools,

the majority supports the free placement of blocks in a two-dimensional space. However, the other 27 tools have other types of placement (e.g., 3D spaces or non-free placement of blocks).

TOOLBAR. The toolbar is where blocks are grouped so that users can look at what language constructs (blocks) are available for further use. Sixty-seven languages have a block-based editor that contains a palette, and 27 do not provide it or it is not explicitly mentioned. Moreover, we analyzed the location of the palette also from the papers. There are four possible locations *top*, *bottom*, *left*, or *right*. We found that 47 tools have the palette on the left-most part of the window. This might be related that the majority of the people read from left to right. Moreover, four tools (*Flip* [118], *Labenah* [6], [206], and *Tuk tuk* [190]) have the palette in the right-most part of the screen. Twelve tools have it in the middle of the screen; this behavior usually presents a stage on one side and the editor on the opposite side. In this way, the palette is in the middle. Finally, *XLBlocks* [149] displays the toolbox at the top of the window and *Tica* [7] does it in the bottom part.

TOOLBAR TYPE. A palette usually groups blocks by categories and this grouping is *static*, meaning users can inspect each category and its blocks, and it will not change. However, we identified that some tools offer a *dynamic* toolbar. A dynamic toolbar is a toolbar that automatically adapts its contents based on the program's current status. In other words, it automatically hides the blocks that cannot be snap into the current status of the program. There are 91 tools that do not support this feature, but *EduBot* [145], *PRIME* [293], and *EUD-MARS* [5] do.

SEARCHABLE TOOLBAR. A *searchable palette* is a palette that has a search bar to help users find blocks without having to open each category. *EduBot* [145] is the only tool that supports a searchable toolbar.

INTERNATIONALIZATION. Given the visual notion of a block-based environment and the possibility of adding descriptions to language constructs in natural language, we investigated if the block-based tools come with support for different languages, which means, if the description of a block can be shown in several languages (e.g., English, Spanish, Dutch). We found that only six tools come with internationalization capabilities, and the vast majority (89) do not support it.

SPRITE-BASED. Sprites are graphic elements of a computer program that can be manipulated as single units. This concept is popular among block-based environment because Scratch supports it. However, we found that is not true for all languages that offer a block-based editor. We identified six tools that support first-class sprites, while the remaining 89 do not.

MOUSE-BASED MANIPULATION. This feature is to reflect how users can manipulate blocks within a block-based environment. Sixty-seven tools support the direct manipulation of blocks using the mouse, while the other 28 tools have different manipulation mechanisms (e.g., physical manipulation).

Summary RQ 5.1

- The feature diagram (Figures 6.3 and 6.4) displays the most important features across block-based environments. There are features at two different levels, *platform*, and *editor*. At the *platform* level, we find features such as documentation, collaborative support, deployment, and stage. The *editor*-level features are the canvas, toolbar, internationalization, and sprite-based editing. Based on our data, we present quantitative analysis to illustrate which tools support each feature. Likewise, we also illustrate the position in which some of these features appear in a block-based environment (e.g., canvas, toolbar, and stage location).
- We identified that –due to the diverse applications in which these environments are used– a standardized set of block-based editor features is missing. Therefore, we propose a feature diagram that summarizes them across different platforms. Notably, we identified two main types of features: platform-based and editor-based.
- We identified that most block-based environments provide a palette that contains all the language construct and a canvas, in which users develop their programs. The stage is a key component in popular platforms, however, their presence varies depending on the language's goal.
- There are two main types of block-based editors: sprite-based (e.g., Scratch) and non-sprite-based.

6.3.3   *RQ 5.2: What are the tools used to develop block-based environments?*

We want to learn how block-based environments are developed. However, given the nature of the papers, this is a non-trivial activity because in most cases we noticed that authors do not mention these details. Below, we present the data we extracted. Depending on how the language was implemented, we classified each paper into one of four categories General-Purpose Programming Language (GPL), grammar, DSL, and not available (N/A). As shown in Table 6.6, 93 tools did not explicitly mention the tools used for its development, 55 were implemented using a GPL, and from the remaining three: one used a visual language, one used a grammar, and one used a DSL, respectively.

Likewise, we studied what programming languages were used in the implementation of these block-based environments. Table 6.7 presents a summary of our findings. For conciseness we grouped some of the languages (for the full list see Appendix B.6). For instance, some languages only mention the use of HTML, so we count it as part of *HTML, JavaScript, and CSS*.

As mentioned before, implementation details are not always discussed, and this is reflected in Table 6.7; 100 papers do not mention what programming language was used for the development. After this, we see that the most popular programming language for the development of block-based environments is JavaScript. Counting all the appearances, this language was used in the development of more than 30 block-based editors. Another interesting fact is that there is only one language developed using a Language Workbench (JastAdd [339]).

Category	# Languages
N/A	93
GPL	55
DSL	1
Visual (blocks)	1
Grammar	1

Table 6.6: Type of languages used to develop block-based environments.

Programming language	Papers
N/A	100
JavaScript	15
HTML, JavaScript, and CSS	15
Java	3
Python	2
iOS (Swift)	2
JavaScript & Java	2
Pharo Smalltalk	2
TypeScript	2

Table 6.7: Programming languages used to implement block-based environments.

Following this direction, we explored whether the papers did not mention programming languages at all, or it was just that they did not present implementation details of their tooling. We used the list of the 50 most popular languages as reported by the TIOBE index [61], but "visual basic" was omitted from the search because of the many false positives with the common words "visual" and "basic". In fact we did not find any block-based editor that was implemented in Visual Basic.

Based on the list of programming languages, we developed a tool [221] for mining the corpus of PDF files and counting the occurrences of each programming language. The results in Figure 6.5 show the popularity of each of programming language. The complete list of details of each language and the number of papers that mention the language is presented in Appendix B.6.

As shown in Figure 6.5, Scratch is by far the language most mentioned across the papers. The reason for this is that most of the current block-based environments took inspiration from it. Then, we found seven programming languages (C, Java, Go, R, JavaScript, D, and Python) mentioned in more than 20% of the papers. These languages' popularity might be related to the technologies used to develop block-based environments, and the libraries offered to support their development (e.g., Blockly).

In summary, we identified that most of the papers do not present implementation details about their languages and editors. However, based on the papers that present implementation details, we found that most of the authors use GPLs. Concretely, most of the papers that presented such details used HTML, JavaScript, and CSS to implement block-based environments. Likewise, we observed that the programming languages used to develop block-based editors are aligned with the 50 most popular languages as classified in the TIOBE index.
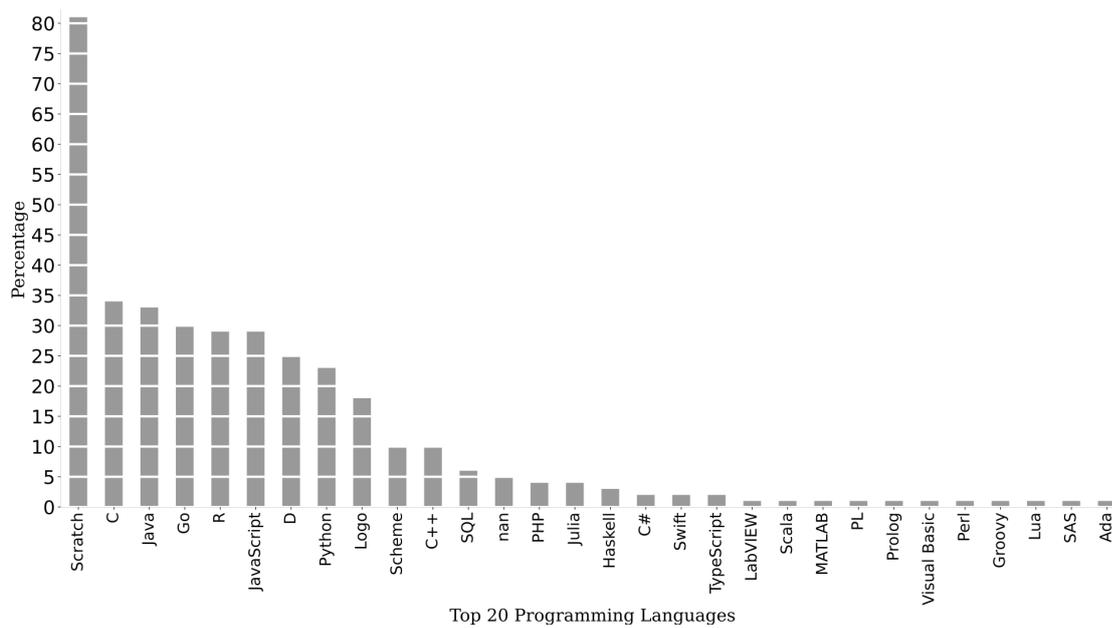
Figure 6.5: Summary of programming languages used for implementing block-based environments.

Summary RQ 5.2

- We identified different ways in which block-based environment are developed. However, most of the authors (93) did not include such details. The most popular way of developing a block-based environment is employing a general-purpose programming language (GPL).
- Since using a GPL is the most common way of developing block-based environment, we identified that the most popular languages for this endeavor are HTML, JavaScript, and CSS.
- Language Workbenches are really under-represented as a means of implementing a block-based editor. There seems to be an opportunity there.

6.3.4   *RQ 5.3: How are block-based environments developed?*

One of the main objectives of this systematic review is to identify how block-based environments are developed in practice. Therefore, we searched the selected papers for the languages and tools used by the authors to develop block-based environments. Based on the data collected (see Table 6.8) we identified two ways of implementing a block-based editor: either authors rely on existing *libraries and frameworks* or they develop them in a *bespoke* fashion. From the corpus of papers that presented a language, tool, or an extension, 88 of them used libraries for the development of their editors, nine papers developed their bespoke editors entirely from scratch, and 54 papers did not

Library	# of languages
N/A	62
Blockly	40
Scratch	11
Snap!	7
Scratch 3.0 (Blockly)	3
CT-Blocks	3
App inventor & Blocky	2
Microsoft MakeCode	2
BlocklyDuino	2

Method	# Languages
Libraries and Frameworks	88
N/A	54
Bespoke	9

Table 6.8: Method used for developing block-based environments.

Table 6.9: List of tools used for the development of block-based editors.

provide a clear insight about how they were implemented, or they did not necessarily introduce a new tool. However, to better understand of how block-based environments are developed, we analyzed the papers to extract the libraries and frameworks used for their development.

Table 6.9 shows a summary of the resulting list of tools and libraries (for the full list see Appendix B.7). Similarly, as identified in Section 6.3.3, most of the papers do not disclose implementation details. However, due to the popularity of some tools for building block-based editors (such as Blockly) in some cases we were able to identify which library was used for their development, even if the authors did not mention them.

As we can observe in Table B.5, there are more than 20 libraries or frameworks used by authors. The most popular tool used for developing block-based environments is Blockly. It is one of the few tools specifically designed to support the development of block-based editors, which explains its observed popularity. Moreover, it is interesting to observe that some of the tools used for building the languages are also block-based environments (e.g., Scratch, Snap!), which means developers rely on existing languages and editors for the development of block-based environments. This is interesting and worth studying in the future, perhaps there is a lack of specialized tools for building block-based environments, or simply the Software Language Engineering (SLE) community is not aware of the opportunities offered by block-based environments.

Summary RQ 5.3

- Most of the block-based editors included in this survey were developed using libraries and frameworks; only nine editors were developed in a bespoke fashion.
- More concretely, the most popular libraries used for developing block-based environments are Blockly and Scratch.

6.3.5 **RQ 5.4***: What languages offer a block-based editor and what are these languages used for?*

As part of this systematic literature review, we sought for the usages of block-based environments. This means, understanding the existing languages that support a block-based editor, and how these languages are being used. While talking to colleagues we noticed that there is a perception that the block-based notation is restricted to computer science education. People also seem to associate block-based environments with children's tools or toys, given their colorful appearance. To check the validity of these perceptions we analyzed the corpus of papers and documented what tools are introduced in each paper and in which fields these tools are used.

The process to extract this data from the papers is described below. First, during the paper review, we collected specific notes in a spreadsheet about each tool. We noted down a possible topic for each tool. Then, with the resulting data, we calculated the number of topics. Initially, we obtained 81 topics, but that classification was not accurate enough to group the papers in a meaningful manner. Thus, to reduce this number and make a more accessible grouping of papers, we defined seven categories: Education, Embedded Computing, Human Computer Interaction, Arts & Creativity, Science, AI, Data Science and Databases, and Software Engineering. This number is significantly lower, and it works appropriately for presenting our findings.

As mentioned previously, we classified the papers into three categories, namely *Research*, *Language*, or *Extension*; and the way we present them in this section differs depending on their type. Research papers are presented with a summary that contains the paper's topic; and *Languages* and *Extensions* are summarized in a table containing the name of the language/extension (*name*), the library used for its development (*metatool*), and its primary usage (*topic*), as shown in Table 6.10.

Below we present each category with a brief description and a table with the papers that belong to it. However, to improve the readability of the current manuscript, some categories have more than one table.

> Category 1 | Education

> This category presents the papers that are mostly related to educational purposes. There is a wide range of applications in which block-based environments are used to teach programming or computer science concepts, and other subjects such as chemistry and mathematics. Likewise, this section presents the importance of block-based environments in educational settings and how the modality (blocks or text) affects the learning process.

Programming literacy

In this category, we grouped several topics and points of view regarding using of the block-based modality in programming education. For instance, Weintrop [376] studied the impact of using a block-based environment in education, and Xinogalos et al. [390] investigates students' perceptions of five popular educational programming environments and the features that introductory programming environments must-have. Similarly, Yoon et al. [391] designed a curriculum that integrates socio-scientific issues in the design and development of mobile apps using App Inventor. Turbak et al. [346] studied the importance of teaching event-based programming in computer science curricula. Dong et al. [94] propose a tinkering environment for students when they struggle in problem-solving activities. Dwyer et al. [96] study the readability of block-based programs by students.

There are different points of view regarding the modality in which programming should be introduced to novice users. Some advocate that visual languages are the best option for introducing novices to programming, while others support text-based languages as the best modality since that is what professional developers use. Thus, researchers have tried to address this topic, and they have work on evaluating the effects that the modality (block-based, text-based, and hybrid) has in the learning process [58, 241, 276, 378, 379]. Franklin et al. [112] study the differences between block-based languages (e.g., Scratch) and text-based languages (e.g., C and Java). Other researchers focus on studying how to ease the transition from a block-based language into a text-based language [186, 381] and the drawbacks users face in this transition [245, 246]. Milne and Ladner [237] study the relevance of accessibility features in block-based environments.

Finally, Table 6.10 presents the tools aimed at teaching computer science concepts in general and learning environments to support the teaching of computational concepts.

Table 6.11 shows the languages used to support and transfer computational skills to learners.

Table 6.12 contains the block-based languages used to teach other subjects such as aerodynamics, Latin language, mathematics, music, and chemistry.

Table 6.13 presents tools aimed to improve the transition from block-based languages to text-based languages, incorporating block-based notation to existing environments such as spreadsheets, languages to support teachers during grading activities, and, finally, languages to support learners with accessibility issues (e.g., hearing impairments).

---

2 This block-based environment is not a complete platform but an extension to an existing system.

Name	Metatool	Topic
MUzECS [23]	Blockly	Explore computer science with a low-cost robot.
RoboScape[204]	NetsBlox	Teach key ideas in computer science using robots.
[368]	MakeCode	Foster computer science education with Lego Mindstorms.
Robot Blockly [375]	Blockly	Programming industrial robot (ABB's Roberta).
HIPE[176]	-	Pedagogy and programming education.
Reduct [12]	-	Gamifying operational semantics.
[313]	Blockly	Introduce young learners to technology with smart homes.
Labenah[6]	-	Learn coding principles via an edutainment application.
Bubbles[285]	Ardublock	Teach programming to children with a robot fish.
Blocks4DS[103]	Blockly	Teach data structures.
Crescendo² [373]	Snap!	Engage students with programming.
[104–106]	Snap!	Add parallel abstractions to block-based languages.
Pirate Plunder [296]	-	Teach abstractions and reduce code smells with a game.
Resource Rush[212]	Blockly	Teach programming in an open-ended game environment.
Block-C[196]	Openblocks	Learn the C language.
Cake [156]	Blockly	Learn the C language.
Block Pictogramming [146]	Blockly	Learn programming with pictograms.
PRIME [293]	Blockly	Learning environment.
Flip [118]	-	Learn computer science in a bimodal environment.
IntelliBlox [336]	Blockly	Introduce programming in game-based environments.
EduBot [145]	Blockly	Learn programming and STEM modules.
Alg-design² [362]	CT-Blocks	Teach algorithmic to novices.
Map-Blocks [366]	CT-Blocks	Teach programming with online weather data.
LAPLAYA [138]	Snap!	Block-based environment for middle school students.

Table 6.10: Languages used to support programming education.

Name	Metatool	Topic
PiBook [63]	Blockly	Transfer computational skills while working on history, biology, and mathematics.
TunePad [124]	Blockly	Introduce computational thinking via sound composition.
C3d.io [213]	Blockly	Create 3D environments to enable STEAM education.
Tuk tuk [190]	-	Teach computational thinking concepts using games.
CT-Blocks [363]	-	Teach computational thinking skills.
ChoiCo [129]	Blockly	Teach computational thinking via modifying games.
[206]	-	Teach computational thinking.
[365]²	Scratch	Teach computational thinking using experiments of fractal geometry.

Table 6.11: Languages used to teach computational thinking skills.

Name	Metatool	Topic
Airblock [57]	Scratch	Teaching programming and aerodynamics.
BlockyTalky [170]	Blockly	Teaching networks.
Ingenium [394]	Blockly	Teaching Latin grammar.
ExManSim [300]	Blockly	Create vignettes for crisis response exercises.
Catrobat [251]	-	Develop mobile applications collaboratively.
MIT App Inventor [266]	Blockly	Develop mobile applications.
EvoBuild[2] [371]	Deltatick[385]	Teach and create agent-based models.
Phenomenological gas particle sandbox[14]	NetTango	Teach agent-based computations through phenomenological programming.
M.M.M. [306]	Blockly	Create an agent-based modeling system to learn science.
[22]	NetTango	Use agent-based modeling for other disciplines (e.g., chemistry).
ScratchMaths[2] [37]	Scratch	Understand mathematical concepts through programming activities.
[188][2]	App inventor	Teach mathematical concepts in primary school.
Tactode[10]	-	Teach math and technology concepts to children.
Sonification Blocks [18]	Blockly	Learn data sonification.

Table 6.12: Languages used to teach subjects other than programming.

Name	Metatool	Topic
Amphibian [48]	Droplet	Enable switching between blocks and text.
Poliglot [202]	Blockly	Smooth transition from blocks to text in education.
HybridPencilCode [9]	PencilCode and Droplet	Transition from block to text notation.
B@ase [333]	Blockly@rduino	Transition from block to text-based environment.
PyBlockly [328]	Blockly	Add a block-based editor for Python.
Stride [187]	-	Add a frame-based editing (blocks and text) to BlueJ.
XLBlocks [149]	Blockly	Add block-based environment for spreadsheets.
NoBug's Snack-Bar[350]	-	Measure students' performance in programming tasks.
GradeSnap[236]	Snap!	Assist teachers in grading block-based projects.
StoryBlocks[191]	-	Teach programming to blind users with a tangible game.
Blocks4All [238]	-	Accessibility support for block-based environments.
[261][2]	Blocks4All	Accessibility support for block-based environments.
Macblockly[2] [66]	Blockly	Block-based support for audiences with disabilities.
[83]	Blockly	Support users with hearing impairments to learn programing.

Table 6.13: Block-based languages applications.

Category 2   Embedded computing

> This category contains all the papers that were associated with some form of embedded computing. This includes languages for programming and manipulation of robots, microcontrollers, and other embedded systems.

Following the idea of embedded computing with a block-based environment, [65] present the benefits of using a block-based language for manipulating and teaching physical components.

Table 6.14 presents all the languages we classified as being part of the embedded computing category. This includes programming robots, embedded systems, Internet of Things (IoT) devices, and controllers.

Name	Metatool	Topic
MakeCode [24]	Blockly	Programming environment for microcontrollers.
NaoBlocks [329]	Blockly	Manipulate Nao robots.
Coblox [318]	Blockly	Programming ABB's industrial robots.
ROS educational[2] [335]	Snap!	Manipulate ROS-enabled platforms.
Robobo [34, 35]	Scratch	Manipulate advanced sensors.
EUD-MARS [5]	Blockly	Use model-driven approach to program robots.
CoBlox [382]	Blockly	Interface for Roberta a single-armed industrial robot.
MakerArcade [315]	MakeCode	Create gaming experiences through physical computing.
UNC++Duino [36]	BlocklyDuino	Program robots to teach CS concepts.
The Coffee Platform [310]	Blockly	Support computational thinking skills through robotics.
LearnBlock [21]	-	Robot-agnostic educational tool.
RP [70]	Blockly	Affordable robot (software and hardware) for education.
mBlock [205]	-	Teach CS and electronics with affordable robots.
CAPIRCI [38]	-	Support collaborative robot programming.
CODAL [90]	Blockly and MakeCode	Create effective and efficient code for embedded systems.
OPEL TDO [185]	Blockly	Test programmable logic controllers by end-users.
Block-based data fusion [51]	-	Define complex event processing pipelines for smart cities.

Table 6.14: Block-based languages in embedded computing.

| Category 3 | Human Computer Interaction (HCI) |

> This category contains papers that focus on a wide variety of aspects of Human-Computer Interaction. We identified aspects such as the usability of block-based environments and their limitations, comparison between different user interfaces (e.g., TUIs, GUIs, and brain-computer interfaces), adding code hints to block-based environment, and supporting end-user development (EUD) through block-based languages.

Most of the papers that fall in this category present a language as summarized in Table 6.15. However, three papers present a more theoretical view. For instance, Holwerda and Hermans [139] present an evaluation to measure the usability of Ardublockly [16], a block-based environment for programming Arduino boards. This evaluation was done using the cognitive dimensions of notations framework [46]. Likewise, Rough and Quigley [298] present the challenges of traditional usability evaluations. Almjally et al. [7] present an empirical study that compares the usage of a block-based language using Tangible User Interfaces (TUIs) and Graphical User Interfaces (GUIs).

Name	Metatool	Topic
Shelves[2] [140]	Blockly	Usability of block-based environment.
[215][2]	iSnap![277]	Improve code hints in block-based environment.
iSnap[2] [278]	Snap!	Add intelligent tutoring system features to Snap!.
[8]	-	Add custom keyboard to block-based languages.
Enrect [330]	-	Introduce noted-link interfaces to represent variables.
Multi-device Grace [312]	Tiled Grace	Support for multi-device environments.
ARcadia [169]	MakeCode	Prototype tangible user interfaces.
VEDILS [248, 249]	App inventor and Blockly	Support end-users to create mobile learning applications with augmented reality.
Jeeves [297]	-	Support end-users to develop applications.
TAPAS [348]	-	Create workflow applications (e.g., IFTTT [141]).
TAPASPlay[2][347]	TAPAS	Support EUD via collaborative game-based learning.
StoryMakAR [115]	BlocklyDuino	Support storytelling with augmented reality and IoT.
Aesop [307]	-	Create digital storytelling experiences.
Neuroblock[76, 77]	Scratch	Build applications driven by neurophysiological data.
NeuroSquare [219]	Blockly	Support brain-computer interfaces using blocks and flow charts.
Neuroflow [137]	Blockly	Block-flow environment for brain-computer interfaces.
Touchstone2 [97]	-	Tool to design HCI experiments.

Table 6.15: Block-based languages in human-computer interaction.

Category 4   Arts & Creativity

This category contains languages used for exploring creativity or as a medium for creating art through block-based constructs or by analyzing users' patterns as a result of their programming activities. Languages that fall in this category are presented in Table 6.16.

Name	Metatool	Topic
Quando [327]	Blockly	Create interactive digital exhibits for gallery visitors.
BlockArt[2] [91]	Scratch	Visualize programming patterns in Scratch.

Table 6.16: Block-based languages in creativity.

Category 5   Science

In this category, we found a single language using the block metaphor for conducting experiments in biology, see Table 6.17.

Name	Metatool	Topic
OpenLH [117]	Blockly	Liquid handling system to conduct live biology experiments.

Table 6.17: Block-based languages in Science.

Category 6   Artificial intelligence, data science, and databases

This section contains block-based languages applied to the domain of artificial intelligence and data science. This includes topics such as machine learning, chatbots, data science topics in general, and databases, as shown in Table 6.18.

Category 7   Software engineering

This category contains different papers that present languages and proceedings that study block-based environments usage in software engineering. Therefore, the reader will find various topics such as code smells in block-based programs, security, testing, refactoring, debugging facilities, and specialized tools for developing block-based languages.

In this category, we have grouped some papers that present a more theoretical view of the application of block-based languages. Hermans and Aivaloglou [136] study code smells in the context of a block-based environment, particularly in Scratch programs, and Techapalokul and Tilevich [338] study code quality in block-based programs using

Name	Metatool	Topic
ScratchThAI[2] [165]	Scratch	Support computational thinking with a chatbot.
SnAIp[2] [151]	Snap!	Enable machine learning within Snap!.
AlpacaML[2] [395]	Scratch 3.0	Test, evaluate, and refine ML models.
BlockPy [75]	Blockly	Introductory programming environment for data science.
Scratch Community Blocks[2] [84]	Scratch	Analysis and visualization of data coming from Scratch.
BlockArt[2] [91]	Scratch	Visualization tool of programming in Scratch.
[107][2]	Scratch 3.0	Engage learners in exploring and making sense of data.
Snap!DSS[2] [128]	Snap!	Allow data stream analyses and visualization.
DBSnap++ [320]	Snap!	Enable specification of dynamic data-driven programs.
DBSnap [319]	-	Build database queries.
BlocklySQL [274]	Blockly	Block-based editor for SQL.
DB-Learn [364]	CT-Blocks	Teach relational algebra concepts.

Table 6.18: Block-based languages in artificial intelligence and data science.

Name	Metatool	Topic
ViSPE [256]	Scratch	Policy editor for XACML.
XACML policy editor [255]	Scratch	XACML policy editor.

Table 6.19: Block-based Languages in security.

a methodology for code smells. Swidan et al. [331] study naming patterns of Scratch programs' variables and procedures following this direction. In contrast, Robles et al. [292] identify software clones in Scratch projects. The usage of static analysis techniques in block-based programs is beneficial, as shown by Jatzlau et al. [150]. They use static analysis techniques of Snap! programs to learn from programmers' behaviors. Likewise, Aivaloglou and Hermans [3] use static analysis techniques to explore Scratch programs' characteristics in software repositories. Finally, Tenorio et al. [343] study different debugging strategies in block-based programs.

Table 6.19 shows two languages that are used in the security domain for defining access control policies. Table 6.20 shows the list of languages used in different topics of software engineering. Based on these tools, we highlight the appearance of one tool, *Processing BBE*, designed specifically for creating block-based environments.

6.3.6 *Block-based Editors Popularity*

So far, we have presented all the block-based languages that we identified in the papers included in this study. As we have seen so far, most of the studies refer to Scratch as

Name	Metatool	Topic
Extension Whisker [323]	-	Testing framework for Scratch
Extension [311]	Blockly	Add block-level debugging features to block-based environment.
Extension [193]	Blockly	Stepwise support for block-based environments.
Processing BBE [194]	-	Create visual block-based domain-specific languages.
Polymorphic Blocks [207]	-	Represent complex structures and visual type information with block-based UI.
LitterBox [111]	Scratch	Detecting bugs in Scratch programs.
QIS [340]	Scratch	Refactoring infrastructure for Scratch.
[339][2]	Scratch 3.0	Automated refactoring tool for Scratch.
Behavioral Blockly [13]	Blockly	Support behavioral programming.

Table 6.20: Block-based Languages in software engineering.

the most popular block-based environment. To verify this, we manually kept track of the occurrences of each tool in each paper. We started with an initial set of block-based languages that we obtained manually from searching at the most popular tools (see Section 6.4). When we had the initial set of languages, we proceeded to read the papers, and in a spreadsheet, we marked when a tool was mentioned and in which paper. As we were reading papers, we added new languages that appeared to the set of block-based languages. It is important to remark that in some cases, papers not only introduced a tool, but they also mention related tools that we also include in the list of tools. This process has an explicit limitation since the discovery of languages is incremental as we read the papers. Therefore, we made a sanity check using the same tool we developed and presented in Section 6.3.3 to mine the corpus of PDF files and collect the occurrences of each tool.

Figure 6.6 shows a summary of the 11 most popular tools (see Appendix B.5 for the full list). Since we used a program to mine the PDFs to double-check our manual results, the tool is not 100% accurate. In Section 6.5, we present some of the limitations of the tool.

As speculated at the beginning, our results show that Scratch is indeed the language most mentioned in all the papers; it was mentioned in more than 80 of the papers of this study. Similarly, Blockly is the second most mentioned language, even though it is not a language but a library for defining block-based languages. The complete list of tools identified in this study and the number of papers in which they appear are shown in Appendix B.5.
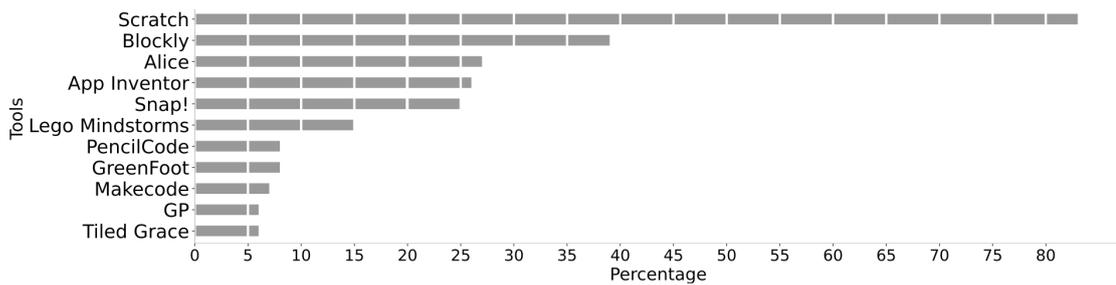
Figure 6.6: Popularity of block-based environments across publications.

6.4   NON-SYSTEMATIC REVIEW OF BLOCK-BASED ENVIRONMENTS

As introduced before, we also conducted a less-systematic exploration using standard Google search to find information about block-based environments that were not necessarily published academically. Table 6.21 presents a summary of our findings after analyzing and trying out each of the tools resulting from the search process. Likewise, it also contains a set of features empirically collected by the first author after testing each language or tool. The process to collect these features was by trying each tool and collecting its features in a spreadsheet. Since all block-based environments do not offer the same features, a few tools had to be tested more than once because some features were included in the spreadsheet after testing the tool. The table is divided into seven columns, and all columns except the first one are subdivided into other columns.

(i) *Name* represents the name of the tool or the language, (ii) *Editor* represents the different components present in a code editor (e.g., mode, error marking, and stage), (iii) *Focus* represents whether the tool is an application, a language that supports the developing block-based environments, or both, (iv) *Deployment* shows the different models in which the tools are being offered, namely as standalone, mobile, or as Software as a Service (SaaS), (v) *Domain* represents the application domain where the tool is used, (vi) *Execution* is how the tool executes an application. We identified mainly two modes: live and pressing an execution button (manual), and (vii) *Licensing* shows the three main types in which the tools are offered.

As the reader might have noticed, these features were used as a basis for the definition of the feature diagram of block-based environments in Section 6.3.2. Using this manual exploration of all available tools we discovered most of the features of block-based environments. The other features were discovered after the systematic literature review process described earlier. As described in the methodology, all tools listed in the table were tested by the first author. Likewise, thanks to the mixed methodology, we were capable of identifying tools that we could not have discovered by relying solely on a systematic approach. Therefore, the less-systematic exploration allowed us to discover 49 tools; from this number, only three tools (*BlockPy*, *CoBlox*, and *Tuk tuk*) appeared in

both the systematic and the less-systematic approach. Some interesting facts of using this mixed methodology are discussed in Section 6.6.

Name	Single mode	Dual mode	Textual	Visual	Hardware	Feedback	Error marking	Application	Language	Standalone	Mobile	SaaS	Gaming	Animation	Programming	Hardware Programming	SE tools	Education	Multi-agent modeling	Live	Manual	Academic	Commercial	Open-source
	Editor		Stage					Focus		Deployment			Domain							Execution		Licensing		
Alice	●			●				●		●			●	●				●			●	●		
App inventor		●		●	●	●	●	●		●		●			●	●		●			●	●		●
Applab		●	●	●		●	●	●				●				●		●			●			●
AutoBlocks	●							●		●														
Blockly	●							●	●	●	●	●	●	●	●	●	●	●						●
BlockPy	●	●	●			●	●	●				●				●		●			●	●		●
Deltatick		●	●					●		●								●	●		●	●		
Frog pond	●			●		●		●				●		●				●			●	●		
GP	●			●				●				●		●		●		●			●			
Greenfoot		●		●				●		●			●			●		●				●		
Hopscotch								●																
Kodika		●			●			●			●				●	●								
Looking Glass	●			●				●		●			●	●				●			●	●		
Makecode	●	●	●					●	●	●		●	●		●	●		●		●		●		
Microblocks	●			●	●			●		●					●	●		●		●				●
Mindstorms	●				●			●		●					●			●			●		●	
miniBloq	●				●			●		●					●			●			●			●
OpenBlocks									●															●
Pencil code		●	●	●		●		●				●		●		●		●			●	●		●
PicoBlocks	●							●		●					●			●			●		●	
Pocket Code	●							●			●		●	●				●			●	●		
Robobuilder	●			●				●		●			●	●				●			●	●		
Scratch	●			●				●				●	●	●				●			●			●
Snap!	●			●				●				●		●				●			●			●
Sphero Sprk	●				●			●		●	●	●			●	●		●			●		●	
StarLogo Nova			●					●																
StarLogo TNG	●			●				●		●				●		●		●				●		
Stencyl		●		●	●	●	●	●		●			●					●					●	
Turtle art	●			●				●		●				●				●			●	●		
Tynker			●	●				●																
Waterbear	●			●				●				●		●		●		●			●			●

Table 6.21: Tools identified using the non systematic approach via standard Google search.

6.5 THREATS TO VALIDITY

A systematic literature review (SLR) is a research methodology used to obtain a complete overview of a particular topic or domain. Based on that, we followed the Kitchenham et al. [178] guidelines, and we defined our protocol for conducting this study. We identified some threats to validity that we discuss in more detail in this section.

6.5.1   *External Validity*

SLRs are conducted to present a summary of a particular topic or domain. Although authors try to reduce their bias as much as possible, it is almost impossible to eradicate it. Thus, this is one of the main threats to validity and a critical aspect of these studies. In the design of our protocol, we tried to minimize as much as possible our bias by defining three filters for including the final set of papers. Moreover, two authors discussed the inclusion and exclusion criteria for a sample of ten papers. Nonetheless, it is essential to mention that since we were looking at specific research questions, this study can never be entirely unbiased, and it is focused on addressing these questions. The queries and the sources of information used in this study prevent us from being fully unbiased. Nonetheless, we tried to keep the current study as broad as possible; in the paper selection, a wide variety of papers came from different communities, venues, and areas of expertise. Moreover, in general, the notion of block-based environments is ambiguous; this term is used to refer to two different topics. On the one hand, visual programming environments that adopt the jigsaw metaphor for creating programs (discussed in this chapter), and on the other hand, the notion of blocks in a block diagram (e.g., Simulink), which is often used in simulation applications and model-based design.

6.5.2   *Internal Validity*

Since the data collection was a manual task, we consider it essential to conduct a sanity check using automated tools. For this purpose, we developed a tool for scanning and mining PDF files and checks whether a given list of words appears in the file's content. There are some known caveats which concern the accuracy and correctness of the tool. First, reading and mining PDF files is not an easy task, mainly because PDF files do not share a standard structure. Thus, some files cannot be opened, or all the text is not parseable. Second, the list of words was manually defined. In the case of programming languages popularity, it was obtained from the TIOBE index [61], which made it more accessible. However, to double-check the languages' popularity, this list was a manual process, which started from a list of languages obtained via a non-systematic method. This list of languages was improved by taking manual notes of new tools presented in papers and their related work. Therefore, it could be the case that the last paper read by the authors introduced a new tool, which of course, was not marked in the previous papers since it was not found yet. However, thanks to the automated tool, we can detect across all the papers if the tool is mentioned or not. In this direction, the tool results are not 100 % accurate due to different factors. (i) *Ambiguous words*. Words in the input list are valid words in English. For instance, *Scratch* or *Go*. Thus, the tool does not differentiate whether it is an English word or refers to a block-based editor or a programming language. (ii) *Punctuation marks*. The tool compares word by word each

of the words in the input list against the text. This means that if a word in the input text appears in the text next to a punctuation mark (e.g., colon or comma), the tool produces a false negative result. The tool says that the word is not present, even though it is present, but it does not capture it since it is next to a punctuation mark (without a blank space in the middle). To measure possible errors in the tool, we sampled ten papers and five programming languages to check how accurate the tool's results are. We calculated type I and type II errors based on the sample to identify the numbers of false positives and false negatives, respectively. The results obtained show that the sensitivity of the tool is 75%. This means that there is a rate of false negatives of 25%. In other words, in 25% of the cases, the tool says that the word is not present in the document, but it is. Similarly, the tool's specificity is about 82,6%, which means that the false-positive rate is 17,4%. In 17% of the cases, the tool said a word was present in the document, even though the word was not present.

In both cases, the tool can be fine-tuned so that both the sensitivity and specificity improve by considering the corner cases previously mentioned. However, that is not the main focus of the current paper. We developed this tool as a sanity check to refine the results obtained during the manual inspection.

In Section 6.3.3, where we present the programming languages used, some papers do not mention how they were implemented. For instance, we could have assumed that when they use Blockly, the editor was implemented using JavaScript, which is the most popular language used for using Blockly. Nevertheless, this is not true for all the cases, because it is also available in other programming languages. Therefore, we decided not to make assumptions about this.

As presented in the protocol, we only considered four academic databases to obtain the academic papers, and the non-systematic search gave us practical languages that do not necessarily have an academic publication. However, the latter means that this part is not easily reproducible.

6.6 DISCUSSION

We identified three main ways that developers follow to create block-based environments. The first approach is by extending an existing language. Twenty-seven of the languages included in this study were developed using this approach. The second one is by using a library that supports the development of such languages. As expected, this is the most popular solution we found in the tools we discovered. Sixty-one languages were developed using other libraries since this reduces the development effort. Finally, the third option is a bespoke implementation. Based on our corpus, only nine languages used this approach. It is important to remark that the previous methods for implementing block-based environments are defined based only on our observations. This might not be true for all cases, given that many authors did not mention any implementation details.

It is interesting to see in the data that there are not many tools that support the whole development cycle of block-based environments. There are specialized libraries for creating concrete pieces of them, but most of these environments rely on code generators. For instance, Blockly is used for describing the UI of the language, and then programs must be compiled to a target language (e.g., Python). We found two tools ([194, 359]) to develop software languages with a built-in block-based editor. However, these two tools are relatively new or not widely adopted; none of the languages presented in this chapter was implemented using them. Likewise, it is relevant to mention that the approach proposed by [194] relies on code generators. Instead, [355] relies on language workbench technology for defining both the syntax and the semantics of languages, which makes such languages also usable outside a block-based editor in a traditional IDE.

Based on the collected data, it is evident that the most popular programming language for implementing block-based environments is JavaScript (Table 6.7). This seems an interesting outlier, but it should not be seen independently from the following observation. Most block-based environments were implemented using Blockly, which is a library implemented in JavaScript. Even though, Blockly offers implementations in other programming languages (e.g., Swift), these have been deprecated and are no longer maintained by the Blockly Team. Moreover, several block-based languages are implemented as web applications, which also explains the vast popularity of using JavaScript for creating block-based languages.

As shown in Section 6.3.4, there is a limited number of libraries for developing block-based environments. Therefore, we see that many authors rely on existing block-based environments to build their own. Surprisingly, specialized language engineering tools (e.g., LWBs) are not used in this domain. JastAdd [262] and ANTLR [263] were used for developing two environments, each one. Our research resulted in *Kogi* [355] (for more details see Chapter 7), that uses the Rascal LWB [180] to create block-based editors for new and existing languages. This to make block-based editors part of the generic services offered by LWBs. However, this tool was not considered in this survey because it was published afterwards.

Another interesting observation that resulted from this study is using mixed methods (systematic and non-systematic searches). As presented in this survey, we see differences between the results obtained from the systematic literature review and the non-systematic tool review. We identified some hypotheses behind these differences. First, some tools are developed to address a specific problem, which is not always followed by a scientific publication. Moreover, there are also industrial applications. Their primary focus is not necessarily the development of scientific publications and following existing literature but to address business requirements and make things work. Another critical aspect of industrial applications is their visibility; sometimes, they are not disclosed due to intellectual property rights. As we underlined in our data, the difference is remarkable. From the 35 languages and tools that we identified in the

non-systematic approach, only 3 had a research paper included in this review. This means that more than 91% of the tools would not have been included if we did not conduct a search of non-academic literature and tools.

6.7 RELATED WORK

Coronado et al. [74] present a literature review about 16 visual programming environments to foster end-user development (EUD) of applications that involve robots with social capabilities. This survey focuses on visual programming environments for non-professional programmers, and they highlight mainly two goals. The first one is to present a list of the tools with their technical features, and the second, to present the open challenges in the development of visual programming environments for end-users. McGill and Decker [217] conducted a systematic literature review and propose a taxonomy for tools, languages, and environments (TLEs) used in computer education. Their main focus is on studying TLEs used in primary, secondary, and post-secondary education. Based on their study, they propose a TLEs taxonomy. Solomon et al. [322] present the history of Logo, a programming environment designed for children to explore mathematical concepts. This is the main predecessor of current notions of block-based environments for end-users.

Rough and Quigley [297] present a perspective of end-user development (EUD) for creating and customizing software by end-users, as end-users outnumbered professional programmers. As a result of their work, they propose some design recommendations to support EUD activities, particularly the creation of software that allows novice users to create apps that collect data (e.g., experience sampling). This chapter follows a similar methodology. They queried computer science databases and a non-systematic approach through Google search to get non-academic tools.

6.8 CONCLUSIONS AND FUTURE WORK

This chapter presents an overview of block-based environments and their features. Also, it presents a detailed view of how these programming environments are developed and the technologies involved in this process. We listed and summarized more than one hundred languages and extensions, which were grouped into seven categories. These categories highlight the fact that block-based environments have a broader scope than computer science education. The results show that authors often do not mention implementation details or possible troubles that the development of a block-based editor has. Moreover, there is a vast diversity of applications in which the block-based metaphor is adopted (e.g., arts, education, science, robotics). Yet, there is a lack of tool support for developing a whole language that supports a block-based editor. Existing tools do not support the whole development cycle of a language. In most cases, designers of block-based environments rely on code generators for defining

the semantics of the languages. We believe that the usage of meta-programming technologies, such as found in Language Workbenches, would enable engineers to fully develop a language and obtain a block-based editor almost "for free", as is the case already for textual editors. Likewise, we confirmed that Scratch has had a significant impact on the development of most of current block-based environments, both conceptually and technically.

Another interesting conclusion of the current survey is that using different methods and sources (systematic and less-systematic, academic and non-academic) allowed us to synthesize a more complete overview of this particular topic than would otherwise be possible. In particular, the less-systematic approach to collect information from non-academic sources presented findings complementary to the systematic literature study, which were also fundamental to the interpretation of the data from the systematic literature study.

We also provided an overview of academic research on usability and learnability of block-based editors (as compared to text editors) and other studies of large collections of block-based programs.

As future work, we foresee different directions: (i) Study what are the best practices for using and implementing block-based editors. The current chapter presents an overview of the features we identified across languages. However, it is interesting to explore the particularities of block-based interfaces to improve the users' programming experience; and how this can be used to implement better block-based editors. (ii) Explore the integration of block-based editors as part of the default set of services offered by specialized tooling for language development (e.g., language workbenches). (iii) Study the lack of tools that support the creation and generation of block-based editors. (iv) Support dynamic aspects of languages in block-based environments (e.g., debugging and live programming). (v) Investigate hybrid environments in which parts of blocks are text-based, depending on the language construct's nature. This to improve the usability and efficiency between drag and drop and writing textual code.

BLOCK-BASED SYNTAX FROM CONTEXT-FREE GRAMMARS

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 chapter 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.

7.1 INTRODUCTION

Block-based environments have received much attention in recent years due to their ease of use for non-programmers [30]. 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:
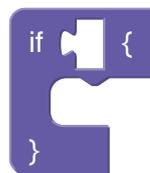
The hole next to the if-level indicates the shape of expressions that are allowed there, and the dent between the curly braces indicates which kind of blocks can be nested under the if-statement. The benefit of such an interface is a what-you-see-is-what-you-get (WYSIWYG) programmer experience and the impossibility of syntax-errors [246, 276, 378, 382]. 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, 73, 266, 375, 382]. They have also been widely investigated as educational tools [65, 77, 138, 145, 196, 333, 376]. 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 [265], Droplet [29], or Open-Blocks [295]. Another way is to extend existing block-based environments like Scratch [288], MIT App Inventor [266], or Snap![243]. Many applications have been developed using these two alternatives. For example, Zhou et al. [394] developed a block-based language for teaching Latin grammar using Blockly. Likewise, Breuch et al. created Airblock [57] 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 chapter, we present an approach, Kogi, to derive block-based languages from declarative context-free grammars, such as used in language workbenches like Rascal [179], Spoofax [166], and Xtext [102]. 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 [119] 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 chapter can be summarized as follows:

- We dissect the structure of block-based environments and model it using an abstract grammar (Section 7.2).

- We present Kogi, a tool that analyzes context-free grammars in Rascal and derives a block-based environment using Blockly's API (Section 7.3). The implementation of Kogi, along with documentation and examples, is available on Github [227].

- We present how the simplification of a context-free grammar impacts the complexity of the generated block-based environment (Section 7.3).

- Kogi's utility is demonstrated by generating block-based interfaces for four languages: State machines, Sonification Blocks, Pico, and QL (Section 7.4).

We conclude this chapter with a discussion of further directions (Section 7.5), related work (Section 7.6), and concluding remarks (Section 7.7).
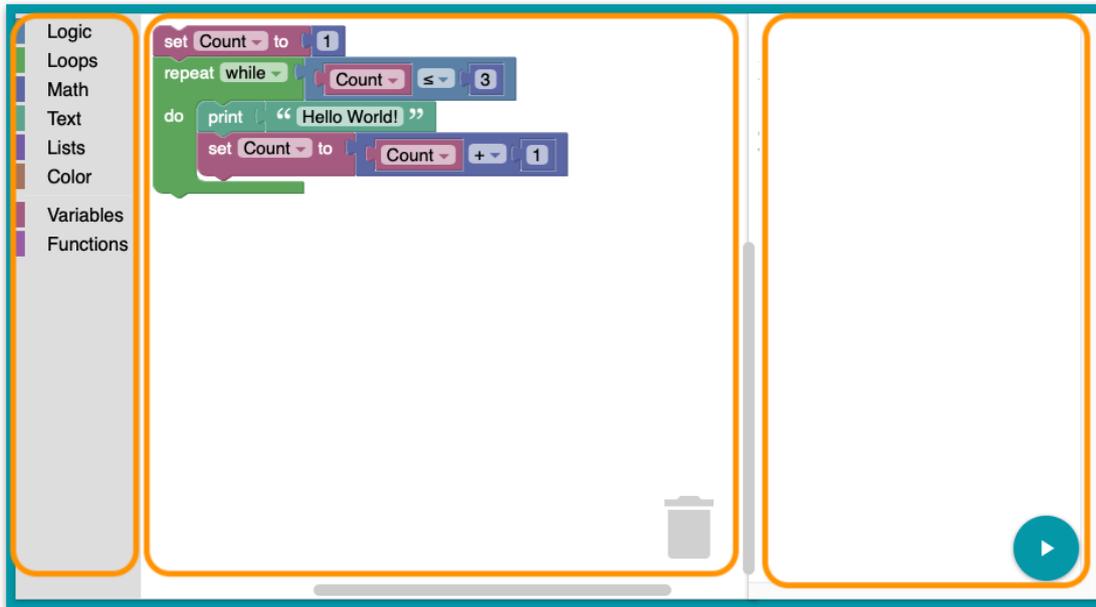
Figure 7.1: Block-based environment built with Blockly.
A block-based environment built with Blockly [119].

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 chapter focuses on block-based environments that adopt the jigsaw metaphor. One of the most known examples of this kind of environment is Scratch [288]. 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 [57], music [18], robotics [382], software engineering [1, 266], arts [327], and biology [117].

Looking at several block-based environments, we split them into three components: a *toolbox*[1], a *canvas*, and a *stage*. It is important to remark that the names of these components vary from one block-based environment to another. Figure 7.1 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.

---

1 The toolbox is also known as palette as mentioned in Chapters 1 and 6.
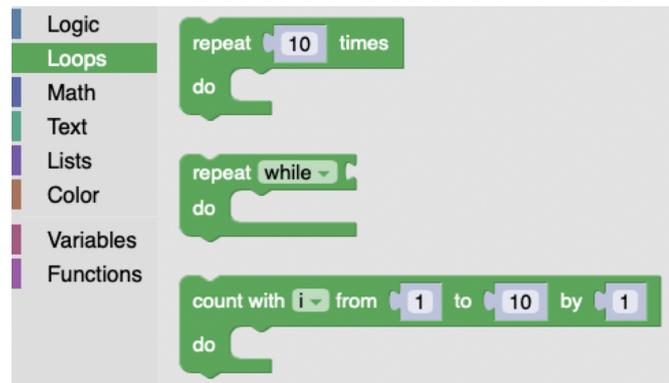
Figure 7.2: Toolbox shelf.
Toolbox shelf [119].

7.2.1 *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 7.1). Each language construct is represented as a block (as the if-statement shown in Section 7.1). 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 [139]. Figure 7.2 shows an example of how one of these shelves look like.

7.2.2 *Canvas*

The canvas (middle view of Figure 7.1) is where the user creates programs (scripts). Block-based programs are created by dragging and snapping blocks together from the toolbox into the canvas [380]. The middle view of Figure 7.1 shows an example of a script created using a block-based language.

7.2.3 *Execution View*

The execution view (right view in Figure 7.1) 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).

7.2.4 *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*.

7.2.4.1 *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 [120]. 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 7.3) inputs and *inline* (condition of the *if* block in Figure 7.4).

Figure 7.3: 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 of a value block. For instance, an *if* block condition (Figure 7.4) 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 7.4 - 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.
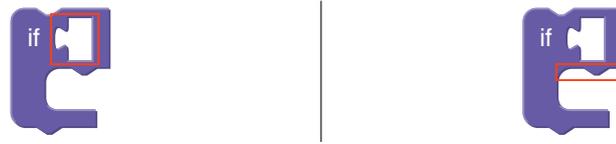
Figure 7.4: If block that highlights the usage of an input value block (left) as its first argument (condition of the *if* statement) and a statement input (right) as its second argument (body of the *if* statement).

7.2.4.2   *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.

7.2.4.3   *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 chapter, 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 7.1.

LEFT OUTPUT.   This connection is visually represented as a male jigsaw connector [121]. 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 7.4 presents an example of a block that supports *previous* and *next* connections.

7.2.5 *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 7.1 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.

7.3 KOGI

Kogi [227] is a tool for describing and deriving block-based environments from context-free grammars using the Rascal [179] 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 7.5 shows a simple DSL grammar for defining state machines, written using Rascal's built-in grammar formalism. It consists of a few rules introducing 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 representation of a type is acquired using the # operator. For instance, consider the following Rascal snippet:

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

Listing 7.1: Algebraic data type modeling Blockly toolboxes.

```
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(lrel[str, 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();
```

```
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]+;
```
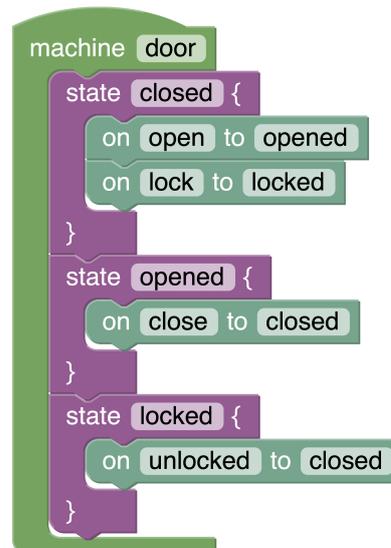
Figure 7.5: State machine grammar (left) and an example state machine (right) using the Kogi generated block-based environment.

The variable typeOfMachine will contain a structured meta-representation of the grammar defined in Figure 7.5, 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.

7.3.1 *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:
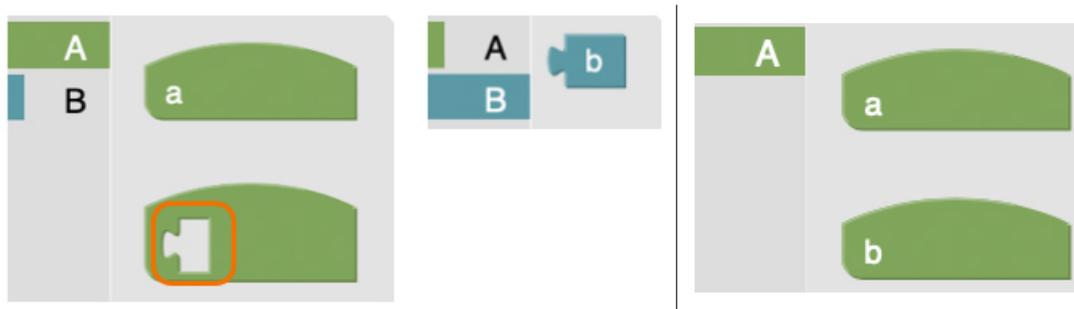
Figure 7.6: Effect of chain rules vs. no chain rules.
Effect of chain rules (left) vs. no chain rules (right).

- 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 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 (p1 and p2), and B has a single production (b). Production p2 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 p2 from A).

The effect is illustrated in Figure 7.6. 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 7.6.

7.3.2 *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 7.5, the mapping of productions to block types is shown in Table 7.1. 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 7.2. 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 7.5, is used to capture identifiers. According to Table 7.2, it is mapped into an inline field of type *Id value*.

Table 7.1: Correspondence between productions and blocks.

Type	Production	Block
Machine	"machine" Id State*	
State	"state" Id "{" Trans* "}"	
Trans	"on" Id "to" Id	
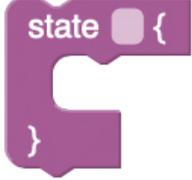
To illustrate the usage of lists within a production rule, consider the State production in Figure 7.5. This production has several literals (state, {, and }, a single lexical element (Id), and a list of transitions Trans). When Kogi finds a list or a separated list[2] it creates a *statement block* with both top and bottom connections. As shown in Table 7.1, both Machine and State blocks allow vertical nesting of states and transitions, respectively.

7.3.3 *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 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.

---

2 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 7.2: 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	
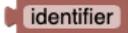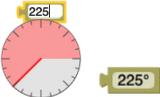
7.3.4 *Execution*

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 7.5 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 7.2 shows an (excerpt) of an XML AST returned by the generated Blockly environment for the state machine language of Figure 7.5. 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.

Listing 7.2: Blockly XML representation of a state.

```
<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>
```

7.4   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 [3]. Below we briefly discuss the latter three languages.

7.4.1   *Sonification Blocks*

Sonification Blocks [18] 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 7.7. Figure 7.8 shows an original Sonification Blocks program compared to the same program in the Kogi-generated environment.

   It can be observed that both programs (Figure 7.8) 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

---

3 https://github.com/cwi-swat/kogi-examples
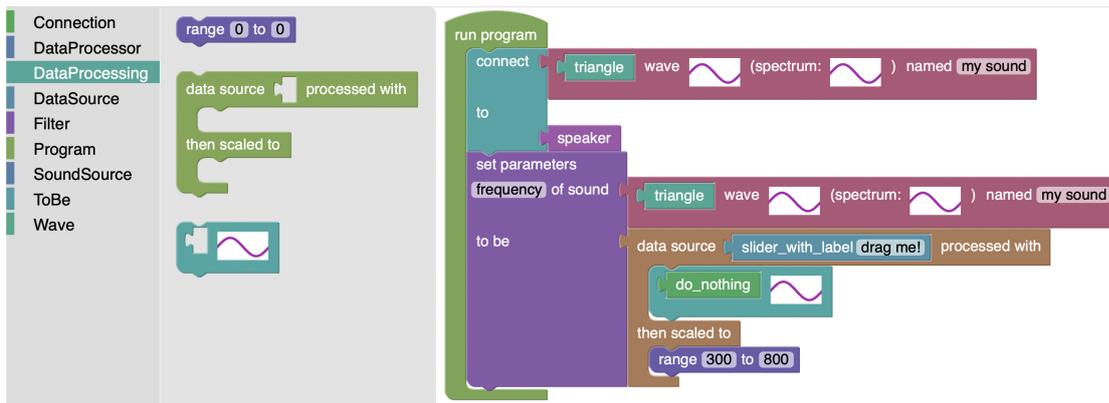
Figure 7.7: Kogi block-based version of Sonification Blocks.

(a)   (b)
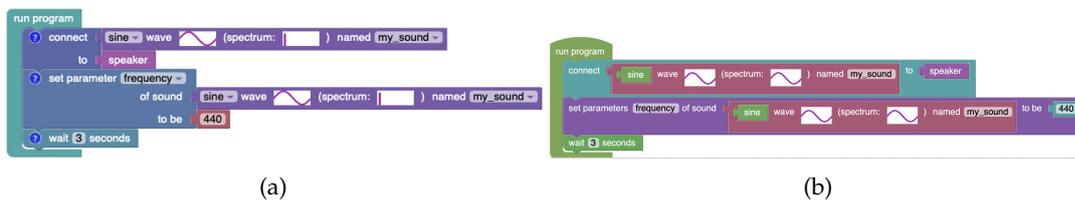
Figure 7.8: Original Sonification Blocks program [17] (a) and Kogi version (b).

category is more significant, since the display of dynamic sine waves is essential for the programmer experience.

As explained in Section 7.3.3, 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 7.3 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 7.9.

7.4.2 *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

Listing 7.3: Customization of Sonification Blocks.

```
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]);
}
```
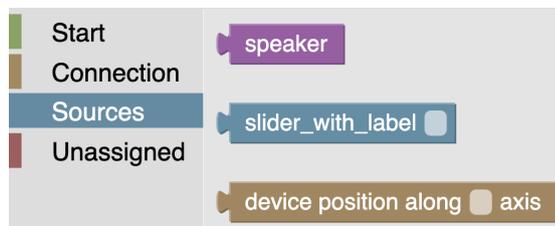

Figure 7.9: Customized version of Sonification Blocks.

of the Rascal standard library [282]. 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 7.10. Kogi allows us to reuse existing language components, not only the grammar for deriving the block-based UI, but the interpreter for executing programs.

7.4.3 *QL*

QL is a DSL for defining interactive questionnaires, and it has been used to benchmark and evaluate language workbenches [99]. 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 [377, 382] 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
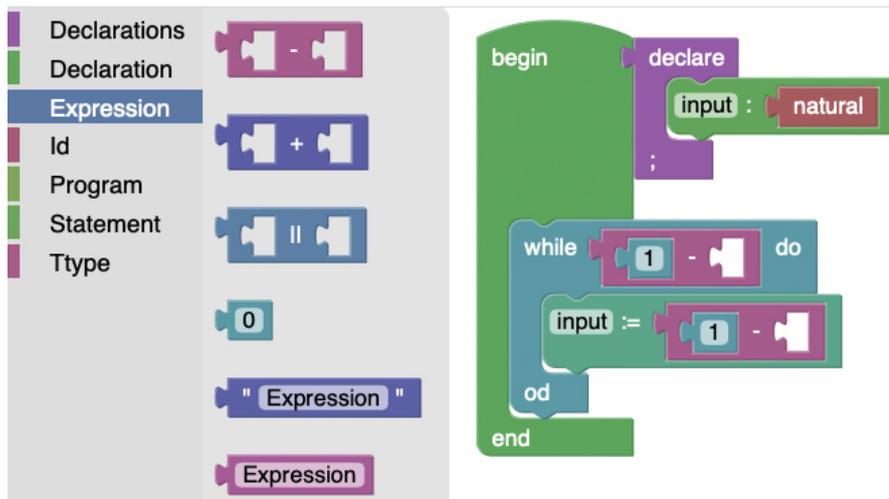
Figure 7.10: Block-based environment for Pico.
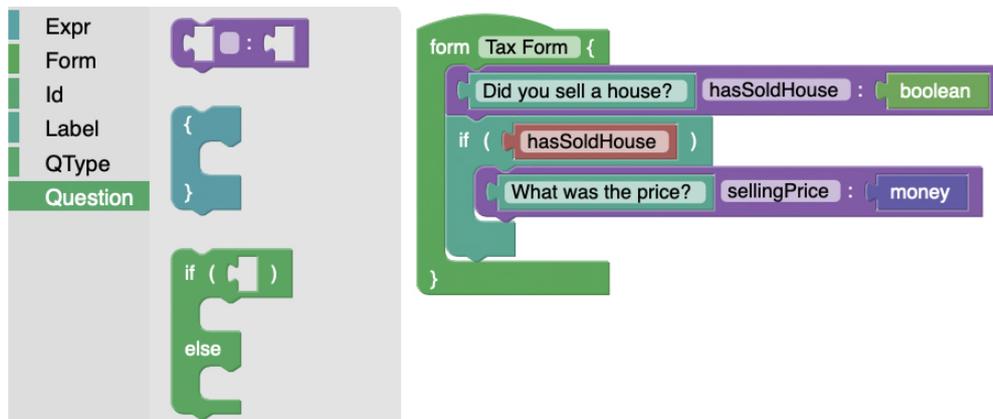
Figure 7.11: Block-based environment for QL.

it as input for Kogi. Figure 7.11 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).

7.4.4 *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 [305] and Cloc [82]. The SLOC for the Rascal grammars were measured only with a configured of Cloc so that it supports Rascal.

Table 7.3: Lines of code (SLOC) 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 7.3 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 [17], 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 7.3, 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,

Table 7.4: 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

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.

7.4.5 *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 7.4 shows the results for all the languages.

In the first row (State machines), as we saw in Table 7.3, there is no difference between the two environments. As we discussed in Section 7.3, 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 7.4, the number of categories decreases in most languages (except in state machines were 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.

7.4.6   *Discussion*

As we explained through the chapter, 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 7.3. For instance, in Sonification Blocks (Section 7.4.1), we used Rascal for describing the language's syntax; based on this definition, we derived a block-based language (Figure 7.8). 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.

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 7.4, 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.

7.5 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 [144, 352] 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 7.2) 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.

7.6   RELATED WORK

Block-based environments can be considered a subclass of the class of graphical or visual languages [74]. One of the main motivations of graphical languages is to make programming for beginners easier than text-based languages [168]. Kogi contributes to the research field of generating programming environments [54, 68, 98, 99, 148, 267, 287]. In the literature, we found mainly two ways of developing block-based environments: through libraries or extending existing block-based environments.

Begel [33] 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. Vallarte [295] designed and developed a framework for creating graphical block programming systems through a specification in XML format. Blockly [265] 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. Tamilias et al. [333] extended Blockly@rduino to create B@SE, a block-based environment to ease the transition from blocks to text-based programming. Similarly, Nergaard [255] created a block-based policy editor for XACML by extending Scratch. Kyfonidis et al. [196] extended OpenBlocks to create a block-based version of the C programming language.

Kurihara et al. [194] 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.

7.7   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 chapter 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 7.2), 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 7.4). The generated environments are evaluated in terms of effort (Section 7.4.4) and toolbox complexity (Section 7.4.5).

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.

GETTING GRAMMARS INTO SHAPE FOR BLOCK-BASED EDITORS

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.

8.1 INTRODUCTION

Block-based environments have become popular thanks to their ease of use, especially for end-users [31]. 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 [246, 276, 378, 383].

An example of a block-based environment is shown in Figure 8.1. It consists of a *palette* (left part of Figure 8.1) that contains all the language constructs that can be used to create programs; the *canvas* (middle-part of Figure 8.1), where users create their programs by dragging and dropping blocks from the palette into the canvas; and an optional *stage* (right part of Figure 8.1) 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) [230].

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 [230].
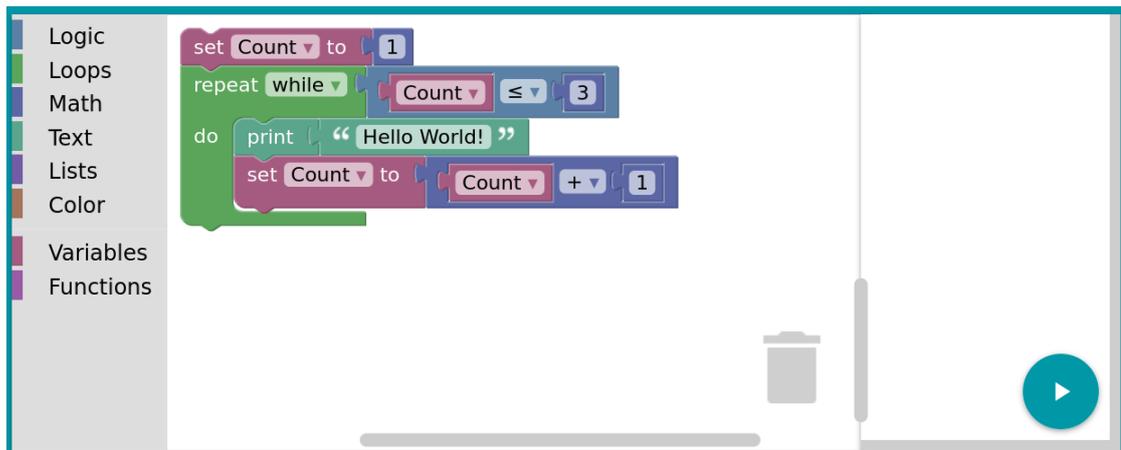
Figure 8.1: Block-based editor created using Google Blockly.

Earlier research presents Kogi (Chapter 7), a tool for deriving block-based editors from declarative Context-free Grammars (CFGs) [355]. 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 chapter, 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 8.2.3.

The contributions of this chapter are:

- an analysis and a set of simplification rules of CFGs to improve the usability of generated block-based editors (Section 8.3).

- an extension of Kogi, S/Kogi [32], that implements the described simplification rules (Section 8.4).

- an evaluation that demonstrates the impact of the simplification rules for deriving block-based editors using six different languages, including Java and JavaScript (Section 8.5).

We continue this chapter with a discussion of limitations and trade-offs, particularly considering the type and complexity of the grammars, as well directions for future work (Section 8.6). Finally, we present related work (Section 8.7), and conclude our chapter (Section 8.8).

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.

8.2.1 *Mapping Top-level Alternatives to Blocks*

As mentioned in Section 8.1, there are three ways to develop block-based editors: implementation from scratch, extending existing block-based editors, or using libraries [230]. An alternative is presented in the work by Verano and van der Storm [355] (Chapter 7), called Kogi, which derives block-based editors from context-free grammars [228]. The resulting block-based editors use Google's Blockly library [119]. 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.

Kogi's mapping from grammars to blocks is based on a set of heuristics [355]. 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 [11, 64]). The implementation of this language is available on GitHub[42].

Table 8.1 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 8.1). 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.

---

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

Table 8.1: Mapping between grammar rules and blocks.

Type	Rule	Block
VarDecl	`Type Id ";"`	

Stmt	`1  "{" Stmt* "}"`
	`2  | "while" "(" Expr ")" Stmt`
	`3  | ...`	

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, 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 [180]. 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.

8.2.2 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 [43]. From this, we will then derive our aesthetic criteria for blocks, following a heuristic evaluation as proposed by Nielsen and Molich [257], 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 chapter, 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 [139], we will apply the Cognitive Dimensions of Notations (CDN) framework [47] 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.
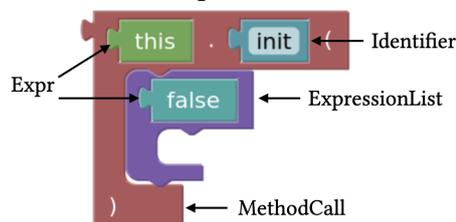
Further, these special-purpose blocks will likely contradict the program structure in which users typically think, leading to higher *diffuseness* of the notation, where more space in the notation is taken up to express a certain construct. As the generated editor also does not communicate the need for 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 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).
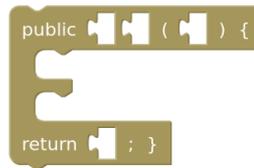
Ambiguous inputs lead to blocks with low *role-expressiveness*, where the purpose of an aspect of the notation is not readily recognizable, and low *visibility* when interacting with the block palette, particularly for novice and end-users.

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.
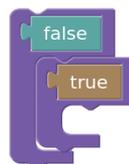
Recursive rules that signify lists have low *role-expressiveness*, as well as low *consistency*, where users apply knowledge from other parts of the notation to new parts, as other

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*.

8.2.3 *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.
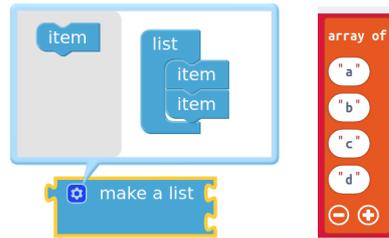
8.3 GRAMMAR SIMPLIFICATION

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 [139] 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.

8.3.1 *Simplification Rules*

Our simplification pipeline consists of four major stages, 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 [154]).

Grammars written in formalisms that do not support left-recursion (e.g., standard PEG [109]), 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" [198, 392].

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 (as shown also in Section 7.3.1).

```
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.

8.3.2 *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 shadow blocks for each block input, as described in Section 8.2.3. 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 8.2.3, 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.

8.4 IMPLEMENTATION

In this section, we present S/Kogi's architecture and selected implementation details. S/Kogi is an extension of the previously described Kogi [355]. Unlike Kogi, which was implemented in the Rascal Language Workbench, S/Kogi is an alternative implementation in Squeak/Smalltalk [143]. 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.
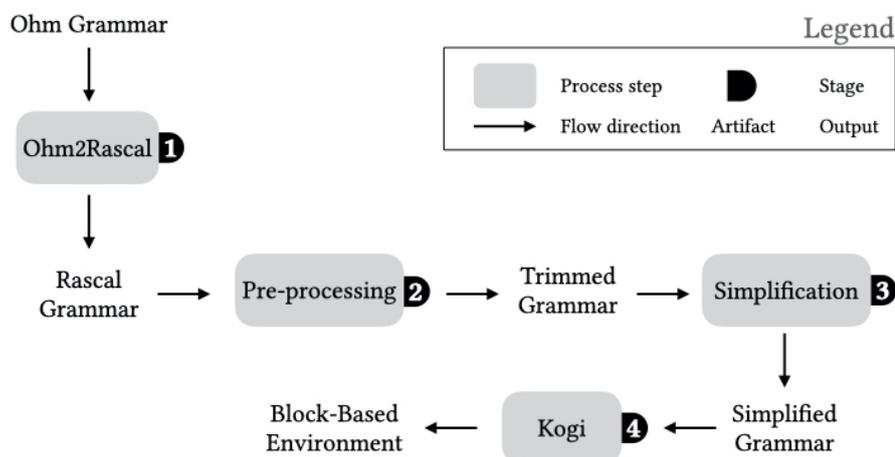
Figure 8.2: S/Kogi's architecture.

Figure 8.2 shows S/Kogi's architecture. While no special features from the grammar dialects are required, their structure tends to differ as outlined in Section 8.3.1. As such, there are two entry points for using S/Kogi: a *Parsing Expression Grammar* (PEG) [108]/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 8.3.1. The simplified grammar (trimmed 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 8.3.2.

Users can apply minor customizations through the grammar's domain objects in Smalltalk code, as discussed in Section 8.5 and Section 8.6. An example of a block generation invocation including customization is shown in Listing 8.1.

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 8.1: Customization for the editor generation process.

8.5 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.

8.5.1 *Simplified MiniJava*

As previous research [233, 355] 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 8.2.3 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.
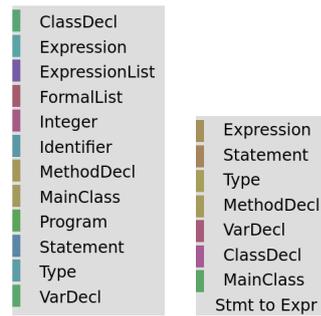
Except for the "Stmt to Expr" category (discussed in Section 8.6.1), all terms found in the S/Kogi palette should appear familiar to Java developers. Exceptions in the Kogi palette are the `FormalList` and `ExpressionList` rules that are both recognized as "simple" rules in S/Kogi and inlined.

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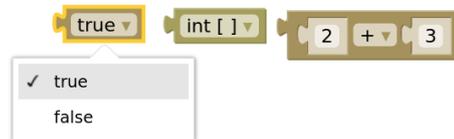
As such, users can quickly change between the related instances of Boolean, type, and operator blocks.

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.
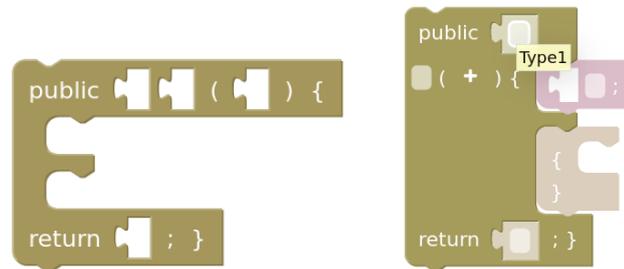
For example, in Kogi's version on the left, users may need to resort to trying various types of blocks to find out that the first statement input is meant for variable declarations and only the second is meant for statements. In S/Kogi's version, the types of blocks are hinted at through the block's 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.
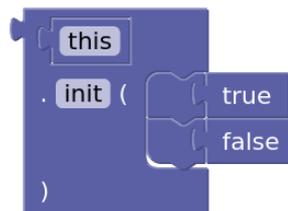
Figure 8.3: A statement input, allowing users to add multiple blocks without the use of recursive blocks using a rubber element as a workaround.

8.5.2 *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 C.1.

CLOUD CONFIGURATION LANGUAGE (CCL).   This language is inspired by the format in Amazon Web Services CloudFormation [314] 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 [99]. QL targets end-users and supports basic control-flow structures. We reused an existing implementation in Rascal [224].

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 [18]. The language's authors heavily customized the block-based editor, for example by adding images of sound waves in blocks.

The grammar [225] 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.
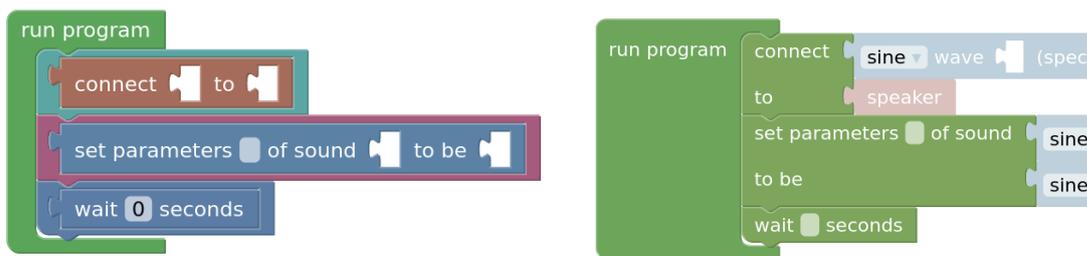
Figure 8.4: On the left, Kogi's output requires placing distinct wrapper blocks for each type of statement. Our simplified version on the right allows using the statements directly.

STATE MACHINE.   This language is a simple DSL for describing state machines [355]. 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 [61] 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 [50] yielded significantly more blocks than the Kogi version, as shown in Table 8.3. 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 EC-MAScript 5, used for teaching source-to-source transformations (desugaring) using Rascal [78], 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 8.1 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.

8.5.3 *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 [305]. 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 8.2 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 8.2: 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

Table 8.3 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

Table 8.3: 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

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 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 C.1.

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 8.2. 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 8.5.2. 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 82% (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 8.3. Overall, as programs written using editors generated by S/Kogi require fewer blocks, as shown in # *Block Prog.* columns (Table 8.3), it is expected that users will find them easier to use. A discussion of this follows in Section 8.6.2. 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.

8.6 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.

8.6.1 *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 8.1. If a block occurs in both contexts, we provide a rubber element as a workaround, seen for example in Figure 8.3. 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.

8.6.2 *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.
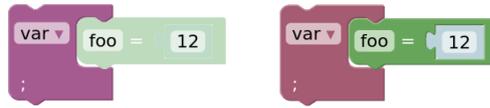
Figure 8.5: On the left, only shadow blocks are used. The expression can only be a number unless users fetch their own assignment block first. On the right, the assignment is already added, but the expression is left as a shadow, allowing the "12" to be replaced by other blocks.

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 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.

8.6.3 *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 8.5. We allow users to specify that this is desirable for certain rules but currently make no attempt to compute this ourselves.

8.6.4 *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.

8.6.5 *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.

8.6.6 *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.

8.6.7 *Usability*

As shown in Section 8.5, 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.

8.7   RELATED WORK

There is a lack of tools that help users to develop block-based environments [230]. 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* [233] follows a similar approach as the one used in *Kogi* [228, 355]; 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 [250], or transformations in the Synthesizer Generator [286]. Some of the simplification rules described in Section 8.3 might provide good results for improving the generated projectional editors as well.

The transformations described in this chapter can be seen in the context of *grammar convergence* [198]. 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 chapter, 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 [384]. This work was further refined in the SDF syntax definition formalism [134], and later in Stratego/XT where algebraic signatures are derived from context-free grammars [155]. In a sense, the block-based definitions derived from grammars in this chapter 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).

8.8 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.

Part V

POSTLUDE

CONCLUSIONS

In this chapter, we revisit the research questions presented in Section 1.2 and summarize the main contributions of our work. Overall this thesis discusses how to engineer and develop language parametric programming environments and user interfaces for Domain-Specific Languages. In particular, we addressed the following central research question:

> **RQ**: *How to engineer different user interfaces for DSLs, so that language engineers can choose the right technological space and notation for various types of users, while reusing existing language components?*

Consequently, eight concrete questions emerged, and we explored them using three technological spaces: computational notebooks, projectional editors, and block-based environments. The first five research questions, **RQ1.1-RQ3**, were answered using the technological space of computational notebooks. Then, **RQ4** was addressed using projectional editors. Finally, **RQ5**, **RQ6**, and **RQ7** were studied from the block-based environment perspective.

In **RQ1.1** and **RQ1.2**, we study the requirements for creating and generating computational notebooks for Domain-Specific Languages (DSLs) and how notebooks can be generated using Language Workbenches (LWBs), respectively.

> **RQ1.1**: *What is required to define a computational notebook at the language abstraction level?*

> **RQ1.2**: *How can notebooks be offered as a generic service in language workbenches?*

We addressed these two research questions in Chapter 2, where we performed a FODA on computational notebook platforms. As a result, we identified notebooks' main characteristics and components and what was needed to create a notebook for a DSL. Moreover, we identified that some of these characteristics are already part of the generic set of IDE services offered by most LWB. Therefore, we developed Bacatá, a language-parametric Jupyter notebook generator for DSLs written using Rascal. Bacatá allows language engineers to create notebooks at the language abstraction level and not at the notebook implementation level, and it helps to generate and develop IDE services for notebooks such as syntax highlighters and auto-completors. Additionally, it allows language engineers to reuse existing language components (e.g., REPLs, interpreters, compilers, and type checkers) to generate Jupyter kernels.

During the development and evaluation of Bacatá (Chapter 2), we identified that some languages were more straightforward to use through a notebook interface than others. Therefore, the following research questions are inspired by this work, and they are closely related to **RQ1.1**and **RQ1.2**. Thus, we studied:

> **RQ2.1**: *What are the requirements for designing a notebook-friendly language?*

> **RQ2.2**: *How can we transform an existing language into a notebook-friendly language?*

In Chapter 3, to characterize the type of languages that are more suitable to be used through a notebook interface, we surveyed and studied different REPLs for some of the most popular languages and the principles underlying those REPLs. Based on this, we identified and defined a class of languages called *sequential languages*. The essence of a sequential language *is that the concatenation of two valid programs is also a valid program*. Thanks to this property, we can develop an *exploring interpreter*, which is a bookkeeping algorithm that stores program states as configurations. This type of interpreter allows users to backtrack to previous executions, a common task in exploratory programming settings. This class of languages naturally fits the REPL metaphor and, therefore, these languages are notebook-friendly. Since not all languages fall into the class of sequential languages nor are notebook-friendly, we introduced a methodology for transforming existing non-sequential languages into sequential languages. As a result, based on the definition of a sequential language, obtaining an exploratory interpreter is straightforward, and this type of interpreter naturally supports the notebook metaphor.

Based on the definition of sequential languages, the development of exploring interpreters for such languages, and the massive usage of computational notebooks for exploratory programming tasks, we defined the following research question:

> **RQ3**: *What language design guidelines could be used to improve the programming experience of DSL users within a notebook platform?*

In Chapter 4 we addressed this questions, by combining Bacatá (Chapter 2) and the principled approach described in Chapter 3. Therefore, thanks to the supported actions in an exploratory interpreter (execute, revert, and display), we explored how to extend the traditional notebook UI and create language parametric widgets that could improve the programming experience of users, mainly when performing exploratory programming tasks. Therefore, we designed two notebook widgets, *execution graph widget* and *variable watcher*. These two widgets offer users additional information about the state of the underlying interpreter, which helps them make better decisions and understand the execution of the notebook. The execution graph widget allows users to understand how the notebook has been executed and change the interpreter's state by backtracking previous results. This is particularly useful for exploratory programming

activities in which users want to try different alternatives within the same notebook. The variable watcher allows users to inspect the name and values of the variables that are currently stored in the underlying notebook interpreter. Inspecting the notebook's interpreter is helpful for end-users because they do not need to keep this information in their mind; they always have an overview of the existing variables. The availability of this information might reduce the users' cognitive load and better understand their programs' output.

After exploring the usage of notebooks as an interface for DSLs, we explored two other user interfaces, projectional editors and block-based environments; they enable different mechanisms to interact with DSLs. First, we explored projectional editors because we worked on a project in collaboration with Canon Production Printing (CPP), and they proposed a concrete use case that involved the usage of projectional editors. The use case that came from CPP is the following: they have developed several DSLs using different technologies, mainly using textual and projectional language workbenches. However, now they must reuse language concepts defined in textual languages within projectional languages to support new business needs. Due to the heterogeneity of the technologies, this task is cumbersome and expensive in terms of effort and money, mainly because languages must be reimplemented from scratch, which is not ideal. Therefore, we defined the following research question:

> **RQ4**: *What mechanism can be used to map context-free grammars to projectional language definitions?*

As presented in Chapter 5, there are some existing approaches from both academia and industry that addressed this problem. However, the existing solutions required re-writing the languages' syntax into a specific formalism. Therefore, we proposed an alternative that relies on the nature of textual languages, which are often described using Context-Free Grammars (CFGs). Therefore, we designed a mapping from CFGs productions into projectional editors constructs. Concretely, we developed a prototype using Rascal as a textual language workbench and JetBrains MPS as a projectional language workbench. Our results show that it is possible to automate the process of deriving an abstract syntax tree and a projectional editor from a CFG, which allows using a textual language within a projectional editor. However, in some cases, resulting projectional editors might require some fine-tuning from language engineers to offer a good editing experience to their users.

After evaluating the approach for generating projectional editors from CFG specification, we explored whether a similar approach could be used for deriving block-based environments from CFGs. However, before diving into the derivation process, we needed to understand these visual programming environments better. Therefore, we first studied block-based editors in general through the following research question:

> **RQ5**: *What are the main characteristics of block-based environments and how are they implemented?*

In Chapter 6, we conducted a systematic literature review and a less-systematic tool review to understand the landscape of block-based environments. We identified that these programming environments are used in different domains beyond the realm of programming education. Regarding the main characteristics of block-based editors, we found that they are heterogeneous, and their features depend on the language, the domain, and the target users. To summarize our findings, we developed a feature diagram with the alternatives we identified. After studying the different tools, we also identified that most block-based editors are being developed in an ad-hoc fashion, and no language engineering technology is being used. Most of the identified tools in this study showed that JavaScript is the most popular language for developing such environments. This finding is not surprising since the most popular library used for developing these environments is Google Blockly, written in such a language. Finally, we found that performing a systematic and a less-systematic review allowed us to obtain more insights about this domain that would not be possible to obtain following a single method.

Based on the findings obtained after the systematic literature review and the less-systematic tool review, we identified that specialized language engineering technology, such as Language Workbenches (LWBs), was not used in developing block-based environments. Therefore, in Chapter 7, we addressed the next research question:

RQ6: *How can language workbenches support the development of block-based environments by reusing existing language components?*

To support the development of block-based editors using language workbenches, we followed a similar approach to the one we used in Chapter 5. Therefore, we start with CFG for describing languages. Then, grammars are analyzed to transform production rules into blocks. To transform productions into blocks, we designed a custom data type to capture the required data to specify block-based editors, including the languages' palette and blocks. For this purpose, we implemented Kogi, which is a tool that takes as input CFGs defined using the Rascal language definition formalism and produces block-based environments that use Google Blockly. In this process, Kogi analyzes the grammars to improve the usability of the resulting block-based editors by inlining chain rules. Although we apply some heuristics to improve the editors; they might still not be ideal out of the box. Thus Kogi offers a mechanism for customizing the generated editors. This mechanism is helpful because it allows language engineers to adapt their generated environments to satisfy their requirements. Moreover, Kogi allows language engineers to reuse existing language components to generate block-based editors and reuse all the language processor components (e.g., REPLs, type checkers, interpreters, and compilers). As a result, block-based programs can be executed using the existing language's infrastructure.

Kogi demonstrated that it is possible to use LWBs to develop block-based environments for existing and new languages. However, some of the resulting block-based

interfaces provided better results than others. Notably, we noticed that DSLs produced better results than programming languages in terms of the aesthetic criteria (e.g., number of blocks and categories) presented in Chapter 8. After generating the block-based editor, some manual work from developers was required to offer a usable programming environment. With this in mind, we defined the following research question:

> **RQ7**: *What techniques could be applied to improve the usability of generated block-based editors?*

In the previous chapter (Chapter 7), we observed that the usability of the generated block-based editors could be improved. Therefore, in Chapter 8, we studied how this can be achieved by applying different heuristics that simplifies the structure of the input grammar. As observed in the results, grammar transformations significantly reduced the number of blocks in the editor and improved their usability. However, a user study is required to determine whether there is a correlation between the reduction in the number of blocks and the increased usability of the block-based editors.

To conclude, this thesis explored different user interfaces and notations for interacting with DSLs such as computational notebooks, notebook widgets, projectional editors, and block-based environments. Moreover, we studied how to engineer these interfaces from the language engineering perspective and maximize the reuse of existing language components. This thesis demonstrates that it is possible to effectively generate different user interfaces for the same language using software language engineering techniques and technologies. Also, as discussed throughout this thesis, there are different benefits of using language engineering technologies and methods rather than ad-hoc implementations in terms of productivity.

Part VI

APPENDIX

A

APPENDIX: BACATÁ: NOTEBOOKS FOR DSLS, ALMOST FOR FREE

A.1 COMPUTATIONAL NOTEBOOK TOOLS

		Features	Zeppelin	Databricks	Burrito	Codestrates	Distill	Graphpad Prism	Colaboratory	Iodide	Jupyter	Knitr	Maple	Mathematica	MATLAB	ObservableHQ	R markdown	Sage
Editor	Editing	Free-form	●	●		●			●	◐	●	●		●	●	●	●	●
		Spreadsheet						●										
	Syntactic services	Highlighting	●	●		●			●	●	●	◐		●	●	●	●	
		Completion	●	●		●			●		●			●	●	●	●	●
		Formatting	●	●		●			●	●	●			●	●	●		
		Folding				●								●	●			
		Keyboard shortcuts	●	●		●		●	●	◐	●			●	●	●	●	
		Line numbers	●	●		●				●				●	●	●		
		Comments				●			●							●		
		Prose	●	●	◐	●	●	●	●	●	●	●		●	●	●	●	
	Code	Monolingual	●	●			●		●		●			●	●	●		
		Polyglot	●			◐			●							●		
		Interactive	●	●		●			●	●			●	●		●		
	Execution	Batch	●	●										●				
		Background		●	●	●							●	●	●			
		Async. tasks		●	●					●						●	●	
		HTML widgets	●	●		●	●		●	●	●	●				●	●	
	Rich output	Plots	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●
		Multimedia	●	●	◐	●	●		●	●	●	●		●	●	●	●	
		Tables				●	●	●	●	●	●			●	●	●	●	
		Plain output		●					●	●		●	●	●		●		
VCS		Cell														●		
		Document	●	●		●			●	◐						●		
Text editor						●					●			●	●		●	
Licensing		Academic				●	●	●	●									
		Open source	●							●	●	●				●	●	●
		Commercial		●					●				●	●	●			
Platform	Deployment	SaaS		●					●	●				●	●	●		
		Standalone	●		●	●	●	●			●	●		●	●			
	Extensibility	3rd party integrations	●	●	●	●			●	●	●					●	●	
		Extensions	●	●	●	●			●	●				●	●	●	●	
	Programming lang. support	Single				●	●	●	●	●			●	●	●	●		●
		Multiple	●	●						●	●	●				●	●	
	Shareability	Offline	●	●		◐		◐	●		●	◐	●	●	●		●	●
		Online	●	●		●			●					●		●		
	Reproducibility	Document	●	●					●		●			●			●	
		Online		●		●			●	●						●	●	

Table A.1: Notebooks features (●, full support; ◐, limited support).

Tool name	URL
Jupyter	https://jupyter.org/ [182]
Knitr	https://yihui.name/knitr/ [218]
Burrito	Research paper [130]
GraphPad Prism	https://www.graphpad.com/scientific-software/prism/
Apache Zeppelin	https://zeppelin.apache.org
Observable	https://observablehq.com/
Iodide	https://alpha.iodide.io/
Distill	https://distill.pub/
Codestrates	Research paper [279]
Maple	https://maplesoft.com/products/Maple/
Azure Databricks	https://azure.microsoft.com/en-us/services/databricks/
Google Colaboratory	https://colab.research.google.com/
R Markdown	https://rmarkdown.rstudio.com/[389]
MATLAB	https://nl.mathworks.com/products/matlab/live-editor.html
Mathematica	https://www.wolfram.com/mathematica/
Sage	http://www.sagemath.org/[324]

Table A.2: List of tools studied for the Feature-oriented domain analysis on computational notebook platforms.

A.2   METAJUPYTERSERVER CLASS

In Bacatá-Core, the MetaJupyterServer class abstracts the communication layer between Jupyter and a language, while the ILanguageProtocol abstracts the language from the tools in a generic way.

PROCESSEXECUTEREQUEST(LISTING A.1 LINE 4) This method is called when an end-user request to execute an input cell. It delegates the evaluation of the input code received as a parameter to the language's REPL.

PROCESSCOMPLETEREQUEST(LISTING A.1 LINE 6) This method is executed when an end-user request to auto-complete a fragment in the current line of code.

PROCESSHISTORY(LISTING A.1 LINE 8) This method returns the list of all the previously executed commands in the current environment.

PROCESSKERNELINFOREQUEST(LISTING A.1 LINE 10) This method is called in the initialization of the language kernel. When an end-user creates a new notebook,

Jupyter calls this method to obtain information about the language (e.g., name, version, language logo, CodeMirror mode).

PROCESSSHUTDOWNREQUEST(LISTING A.1 LINE 12) This method is used to stop a language kernel.

PROCESSISCOMPLETEREQUEST(LISTING A.1 LINE 14) This method decides whether the input cell code is complete or not. If the code is incomplete, it tells the frontend to display a continuation prompt.

MAKEINTERPRETER(LISTING A.1 LINES 16-17) This method returns an instance of the `ILanguageProtocol`. This instance acts as a bridge between the Jupyter's communication layer and the language. Thus, this object interacts with the language's REPL.

```
1  public abstract class JupyterServer {
2    ...
3
4    abstract void processExecuteRequest(ContentExecuteRequest contentExeReq, Message msg);
5
6    abstract Content processCompleteRequest(ContentCompleteRequest contentCompleteRequest);
7
8    abstract void processHistoryRequest(Message msg);
9
10   abstract Content processKernelInfoRequest(Message msg);
11
12   abstract Content processShutdownRequest(ContentShutdownRequest contentShutdownRequest);
13
14   abstract Content processIsCompleteRequest(ContentIsCompleteRequest isCompleteRequest);
15
16   abstract ILanguageProtocol makeInterpreter(String source, String replQualifiedName,
17    String... salixPath) throws Exception;
18 }
```

Listing A.1: Abstract methods of the MetaJupyterServer class.

A.3   ILANGUAGEPROTOCOL INTERFACE

```java
public interface ILanguageProtocol {

  void initialize(Writer stdout, Writer stderr);

  String getPrompt();

  void handleInput(String line, Map<String, InputStream> output, Map<String, String>
    metadata) throws InterruptedException;

  void handleReset(Map<String, InputStream> output, Map<String, String> metadata) throws
    InterruptedException;

  boolean supportsCompletion();

  boolean printSpaceAfterFullCompletion();

  CompletionResult completeFragment(String line, int cursor);

  void cancelRunningCommandRequested();

  void terminateRequested();

  void stackTraceRequested();

  abstract boolean isStatementComplete(String command);

  void stop();

}
```

Listing A.2: ILanguageProtocol interface.

B

APPENDIX: WHAT YOU ALWAYS WANTED TO KNOW BUT COULD NOT FIND ABOUT BBES

B.1 PHASE 2: FILTERING QUESTIONS

- Is the publication a full paper?
- Does the paper introduce a language or a tool that uses a block-based editor?
- Does the paper introduce a tool for building block-based environments?
- Does the paper use or study block-based environments?
- Does the paper present implementation details regarding the block-based environment?
- Does the paper present best practices for using block-based environments?
- Does the paper present best practices or guidelines for implementing block-based environments?
- Does the paper present limitations of block-based environments?
- Does the paper present open challenges that should be addressed with block-based environments?

B.2 PROGRAMMING LANGUAGES POPULARITY

Language	Occurrences		
		PHP	6
		Julia	6
		Haskell	5
Scratch	124	C#	3
C	52	Swift	3
Java	50	TypeScript	3
Go	46	LabVIEW	2
R	44	Scala	2
JavaScript	44	MATLAB	1
D	39	PL	1
Python	35	Prolog	1
Logo	28	Visual Basic	1
Scheme	16	Perl	1
C++	15	Groovy	1
SQL	9	Lua	1
NaN	8	SAS	1
		Ada	1

Table B.1: Programming languages popularity in block-based environments.

B.3   PAPERS PER VENUE

Category	Venue	# Papers
Human computer interaction	Conference on Human Factors in Computing Systems (CHI)	13
	Conference on Interaction Design and Children	11
	International Conference on Human-Computer Interaction (HCII)	4
	International Journal of Child-Computer Interaction	3
	Creativity and Cognition	1
	International Conference on Tangible, Embedded, and Embodied Interaction	1
	IFIP Conference on Human-Computer Interaction (INTERACT)	1
	Symposium on User Interface Software and Technology (UIST)	1
	International Conference of Design, User Experience, and Usability (DUXU)	1
	International Symposium on End User Development	1
	Iberoamerican Workshop on Human-Computer Interaction (HCI-COLLAB)	1
	International Conference on Human Systems Engineering and Design: Future Trends and Applications (IHSED)	1
Programming / HCI	Blocks and Beyond Workshop (B&B)	13
	Symposium on Visual Languages and Human-Centric Computing (VL/HCC)	9
	Journal of Visual Languages & Computing	2
	International Symposium on End User Development (IS-EUD)	2
Distributed computing	International Conference on Computing, Communication and Networking Technologies (ICCCNT)	5
	Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies	1
	International Smart Cities Conference (ISC2)	1
	International Parallel and Distributed Processing Symposium Workshops (IPDPSW)	1
	Journal of Parallel and Distributed Computing	1
Robotics /Education	International Conference on Robotics and Education (RiE)	4
	Robotics in Education	1
Programming	Science of Computer Programming	2
	Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Splash-E)	1
Security	International Conference on Information Systems Security and Privacy (ICISSP)	2

Table B.2 continued from previous page

	Technical Symposium on Computer Science Education (SIGSE)	10
	Global Engineering Education Conference (EDUCON)	5
	Computational Thinking Education	5
	Education and Information Technologies	2
	Workshop in Primary and Secondary Computing Education	2
	International Conference on International Computing Education Research	2
	Conference on International Computing Education Research	2
	Transactions on Computing Education	1
	International Conference on Information Technology Based Higher Education and Training (ITHET)	1
	International Conference on Learning and Teaching in Computing and Engineering (LaTICE)	1
Education	Journal of Computing Sciences in Colleges	2
	Innovation and Technology in Computer Science Education (ITiCSE)	1
	Workshop in Primary and Secondary Computing Education (WIPSCE)	1
	Conference on Innovation and Technology in Computer Science Education	1
	International Journal of Artificial Intelligence in Education	1
	International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP)	1
	Computers & Education	1
	International Conference on Artificial Intelligence in Education (AIED)	1
	Journal of Science Education and Technology	1
	International Conference on Blended Learning (ICBL)	1
	Conference on Computer Supported Education (CSEDU)	1
	International Conference on Interactive Collaborative Learning (ICL)	1
	Conference on Learning and Collaboration Technologies (LCT)	1
	Software Data Engineering for Network eLearning Environments	1
Accessibility /HCI	Conference on Computers and Accessibility (ASSETS)	2
	Conference on Source Code Analysis and Manipulation (SCAM)	2
	Conference on Program Comprehension (ICPC)	1
	Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	1
Software engineering	Conference of the Center for Advanced Studies on Collaborative Research (CASCON)	1
	IEEE International Workshop on Software Clones (IWSC)	1
	IEEE Transactions on Emerging Topics in Computing	1
	Conference on Soft Computing and Software Engineering (SCSE'15)	1
	Journal of Systems Architecture	1

Table B.2 continued from previous page

	Conference on Open Systems (ICOS)	1
Technology	Conference on Advances in Information Technology	1
General	Communications of the ACM	1
	IEEE Access	1
	TechTrends	1
	Science and Information Conference (SAI)	1
Engineering	Conference on Automation, Computational and Technology Management (ICACTM)	1
	Conference on Developments in eSystems Engineering (DeSE)	1
	Computers & Electrical Engineering	1
Robotics	Iberian Robotics conference (Robot)	1
	International Conference on Ubiquitous Robots (UR)	1
Games	International Conference on Serious Games, Interaction, and Simulation (SGAMES)	1
Multimedia	Multimedia Tools and Applications	1

B.4 PAPERS PER COUNTRY
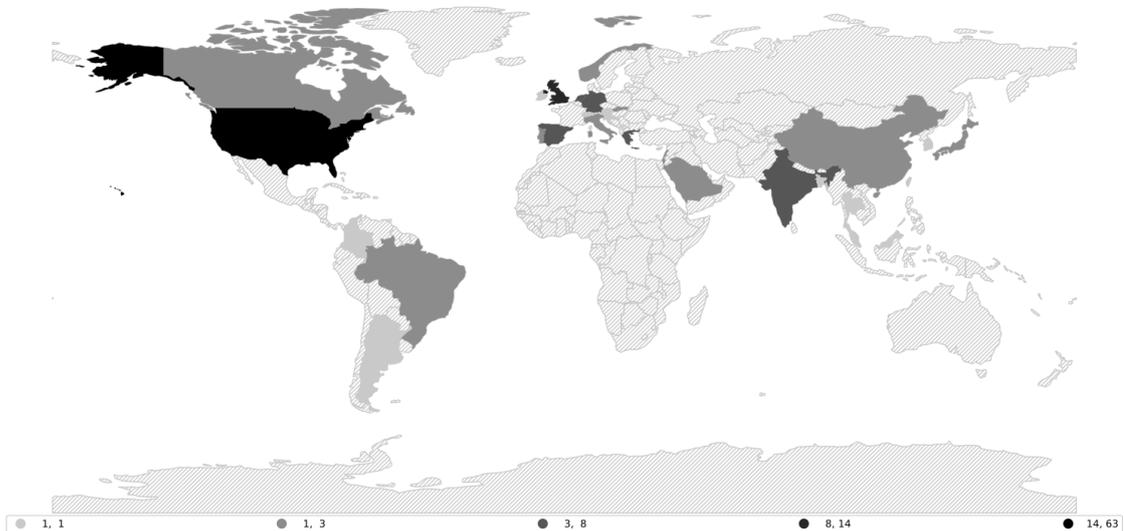
Figure B.1: This choropleth map displays a summary of the number of papers published per country. The detailed table with the values is shown in Table B.3.

Country	Papers
USA	64
UK	14
Germany	8
Spain	6
India	6
Greece	5
Netherlands	5
New Zealand	4
Portugal	3
Brazil	3
Norway	3
Japan	3
Israel	3
Canada	2
China	2
Saudi Arabia	2
Lebanon	2

Italy	2
Slovakia	2
Thailand	1
Ireland	1
Malaysia	1
Scotland	1
Taiwan	1
Slovenia	1
Bangladesh	1
Argentina	1
Colombia	1
Croatia	1
South Korea	1
Korea	1
Austria	1
Switzerland	1

Table B.3: Total number of papers included in this study per country

B.5   TOOL POPULARITY ACROSS PUBLICATIONS

Name	Sum
Scratch	127
Blockly	59
Alice	42
Snap!	40
App Inventor	40
Lego Mindstorms	23
PencilCode	13
GreenFoot	12
Makecode	11
GP	9
Tiled Grace	9
Dr. Scratch	6
Code.org	6
Scratch Jr	6
Droplet	5
LogoBlocks	5
PocketCode	5
Ardublockly	5
StarLogo TNG	5
BYOB	4
NetsBlox	4
Nettango	4
Blocks4All	4
Stencyl	4
CT-Blocks	4
Blocklyduino	4
BlockyTalky	4
Hairball	4
RoboBuilder	3
pencil.cc	3
Deltatick	3
DrawBridge	3
Openblocks	3
ArduBlock	3
Modkit	3
Squeak eToys.	3
Bags	3
Dash and dot	3
Open roberta	3
Hopscotch	3
CoBlox	2

BridgeTalk	2
Tern	2
Ozobots	2
Waterbear	2
Torino	2
Agentcubes	2
mBlock	2
EvoBuild	2
UML	2
DataSnap	2
Ladder	2
Kodu	2
Blockly@rduino	2
Tynker	2
Frog Pond	2
iSnap	2
LaPlaya	1
ITCH	1
DBSnap	1
BEESM	1
CustomPrograms.1	1
CodeIt	1
CARMEN	1
Pixly	1
BlockImpress	1
Polymorphic Blocks	1
ecraft2learn	1
Spherly	1
FabCode	1
MakerArcade	1
MORPHA	1
RoboBlockly	1
edbot	1
NEPO	1
Patch	1
DStBlocks	1
Neuroblock	1
ROBOLAB	1
Amphibian	1
GradeSnap	1
Accessible Blockly	1
PopBots	1
Quality Hound	1

Calico Jigsaw	1
BlueJ	1
SPARQL	1
Tickle	1
ScratchX	1
Flip	1
Logo	1
Code3	1
Snap4Arduino	1
E-Block	1
RoboBlocks	1
Microsoft Touch Develop	1
MicroApp	1
Blockpy	1
AppLap	1
CustomPrograms	1
Robobo	1
CodeSpells	1
Turtle Art	1
Micropython	1
Sketchware	1
Thunkable	1
AppyBuilder	1
BLOX	1
Finch robot	1
App Inventor Java Bridge	1
Robokol	1
BehaviourComposer	1
AgentSheets	1
TurtleArt	1
PicoBlocks	1
Scratch Memories	1
Scratch Community Blocks	1
PseudoBlocks	1
OzoBlockly	1
TinkerBlocks	1
ViMAP	1
KidSim	1
Gameblox	1
Phratch	1
Romo	1
N-Bot	1
Cherps	1

B.6 PROGRAMMING LANGUAGES USED TO IMPLEMENT BBES

Programming language	Papers		
N/A	100	AngularJs and TypeScript	1
JavaScript	15	Snap!	1
HTML, JavaScript, and CSS	15	Python, HTML, CSS and Javascript	1
Java	3	Java / JastAdd	1
Python	2	Java / ANTLR	1
iOS (Swift)	2	ActionScript	1
JavaScript & Java	2	PHP-based framework & JavaScript and HTML5	1
Pharo Smalltalk	2	Tiled Grace	1
TypeScript	2		

Table B.4: Programming languages used to implement block-based environments.

B.7 TOOLS USED FOR DEVELOPMENT

Library	# of languages		
NaN	62	Blockly, Monaco, MakeCode	1
Blockly	40	iSnap [277]	1
Scratch	11	Droplet	1
Snap!	7	MakeCode	1
Scratch 3.0 (Blockly)	3	Snap! & Scratch	1
CT-Blocks	3	Deltatick	1
App inventor & Blocky	2	NetsBlox	1
Microsoft MakeCode	2	PencilCode, Droplet	1
BlocklyDuino	2	Openblocks [294]	1
Python	1	Ardublock	1
NetTango	1	Tiled Grace	1
Blockly@rduino	1	Scratch & Blockly	1
Blockly and Standard pictogramming	1	Snap! & DB Snap	1
Nettango	1	App inventor	1

Table B.5: List of tools used for the development of block-based editors.

B.8   DOMAINS

Sub-category	Category
End-user programming	
Parallel Programming	
Behavioural programming	
Audio programming	Programming
Collaborative Programming	
Programming	
Programming Models	
Education	
Parallel Programming education	
Computational Thinking	
Programming education	
socioscientific issues (SSI) education	
Grammar education	Education
Education programming	
Computer Science Education	
Simulation-based training	
Latin	
Parson problems	
Programming environments	
Novice programming environments	
Accessible Programming Environments	
Spreadsheets	Programming environments
Block-based Environments	
Frame-based editing	
Security	Security
Privacy	
Robotics	
Micro:bit	
Collaborative robots	
Embedded systems development	
Physical computing	Physical computing
Microcontrollers	
Automotive manufactoring	
Smart cities	

Sub-category	Category
Mobile	
Scratch	
Visual DSLs	Languages
Block-based languages	
Augmented reality	
Human Robotics Interaction	
Usability	
Tangible surfaces	
Intelligent Tutoring Systems	
HCI	HCI
Brain-Computer Interface	
Tangible user interfaces	
Brain-computer Interfaces	
end-user development	
Accessibility	
Visualization	
Gaming	
Music	
Cultural Heritage (museums & art)	ARTS & Creativity
Tinkering	
Storytelling	
Biology	
Chemistry	Science
Medicine	
AI	
Data Science	
Data-driven programs	
Data Analysis	AI
data analysis and visualization	
Machine learning	
program analysis	
Code quality	
Testing	
Databases	Software Engineer
Static analysis	
program analysis and transformation	

Sub-category	Category
Software quality	
Agent-based computational model	
Agent-based modelling	Agent-based model
Agent-based modelling tasks	

C

APPENDIX: GETTING GRAMMARS INTO SHAPE FOR BLOCK-BASED EDITORS

C.1 EXAMPLE PROGRAMS

This appendix contain screenshots of the example programs developed in Section 8.5.
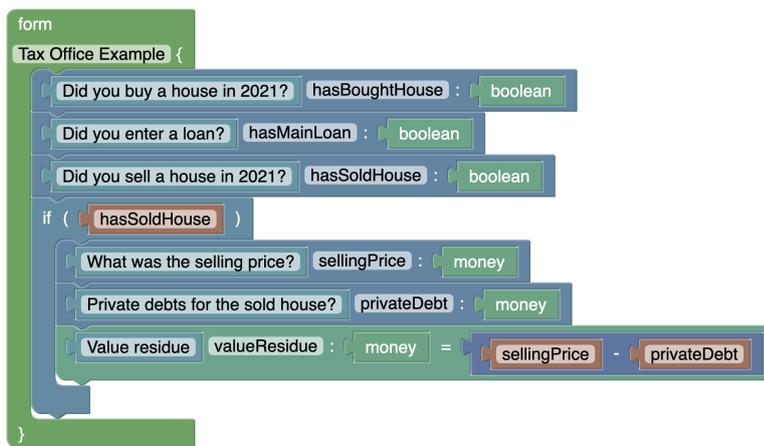
Figure C.1: Example program using the QL environment generated by Kogi.
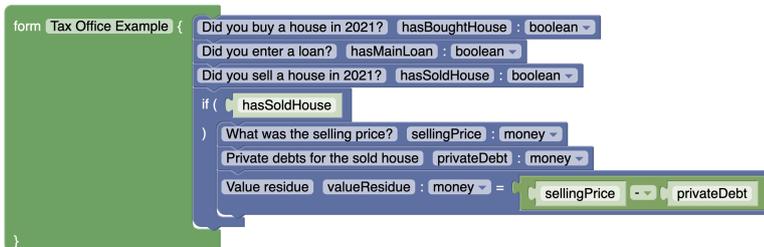
Figure C.2: Example program using the QL environment generated by S/Kogi.
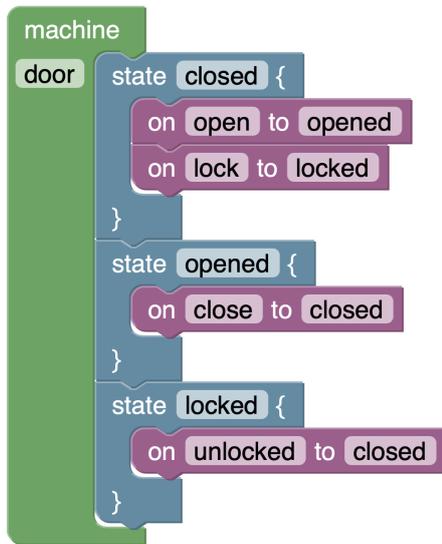
Figure C.3: Example program using the State Machine environment generated by Kogi.
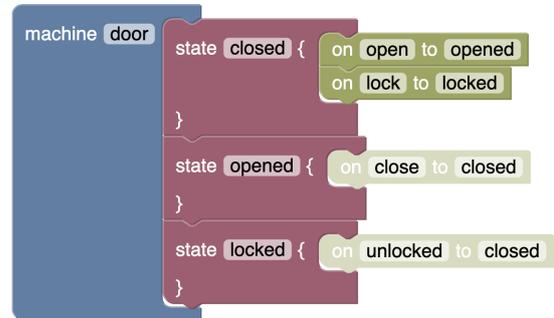
Figure C.4: Example program using the State Machine environment generated by S/Kogi.
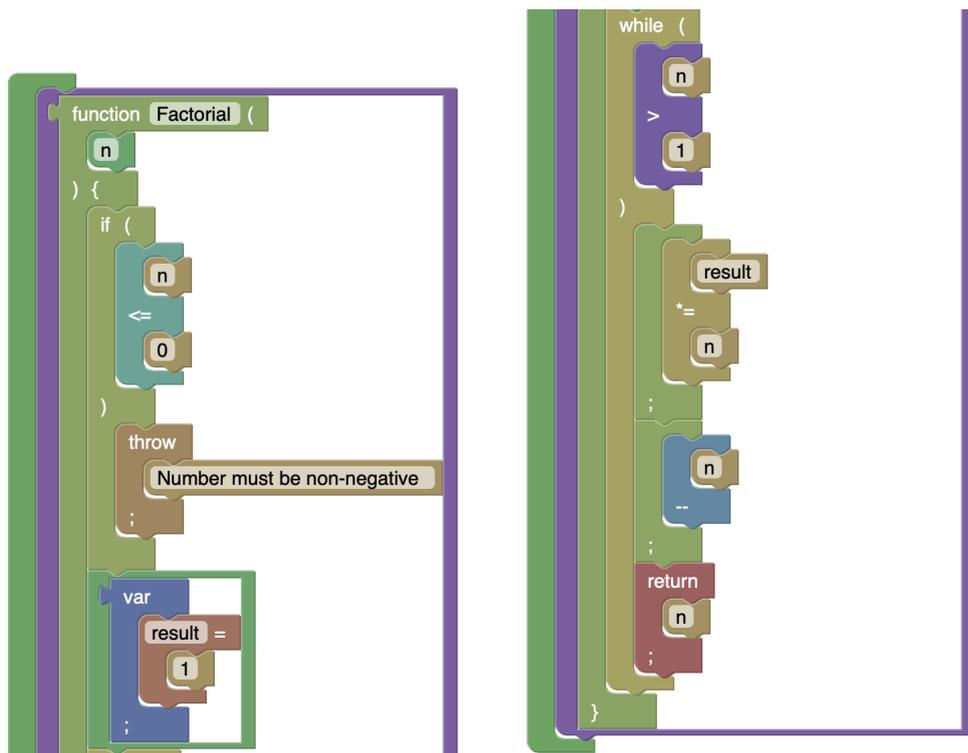
Figure C.5: Example program using the JavaScript environment generated by Kogi.
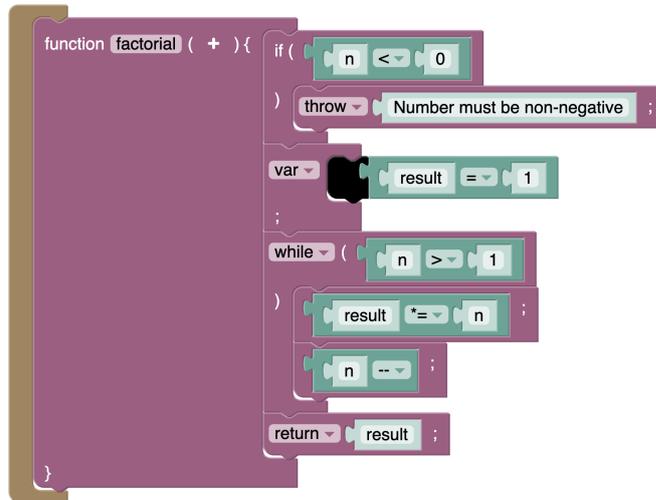
Figure C.6: Example program using the JavaScript environment generated by S/Kogi.
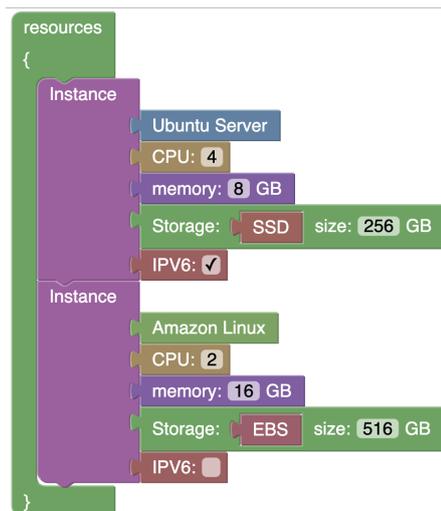
Figure C.7: Example program using the CCL environment generated by Kogi.
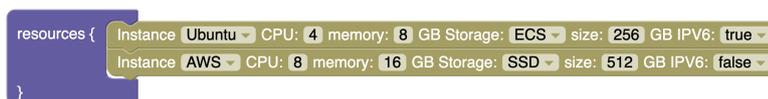
Figure C.8: Example program using the CCL environment generated by S/Kogi.
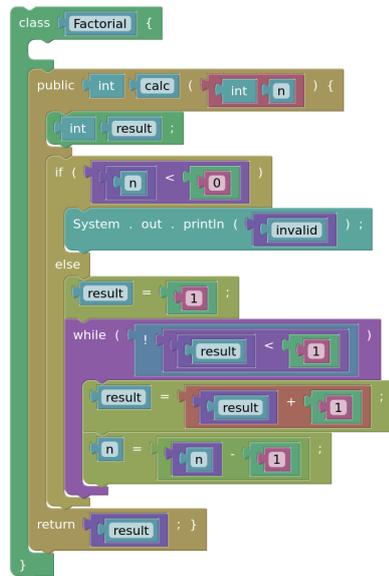
Figure C.9: Example program using the Mini-Java environment generated by Kogi.
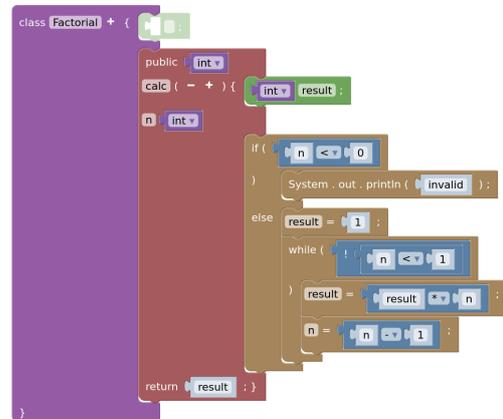
Figure C.10: Example program using the Mini-Java environment generated by S/Kogi.
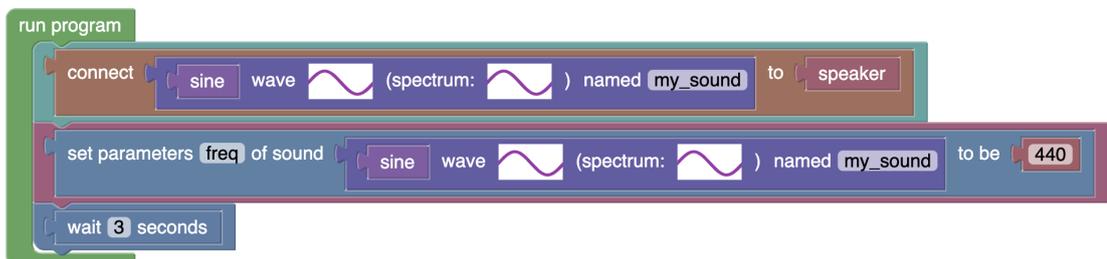
Figure C.11: Example program using the Sonification blocks environment generated by Kogi.
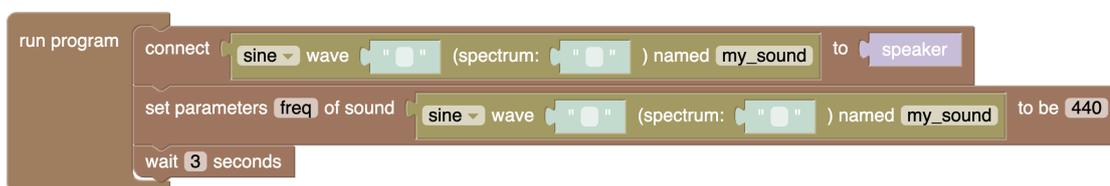
Figure C.12: Example program using the Sonification blocks environment generated by S/Kogi.

BIBLIOGRAPHY

[1] Adaptavist. *AutoBlocks for Jira*. 2019. URL: https://bit.ly/31ioKx5 (visited on 07/08/2020).

[2] Saksham Aggarwal, David Anthony Baau, and David Bau. "A blocks-based editor for HTML code." In: *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015* (2015), pp. 83–85. DOI: 10.1109/BLOCKS.2015.7369008.

[3] Efthimia Aivaloglou and Felienne Hermans. "How Kids Code and How We Know: An Exploratory Study on the Scratch Repository." In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ICER '16. Melbourne, VIC, Australia: ACM, 2016, pp. 53–61. ISBN: 978-1-4503-4449-4. DOI: 10.1145/2960310.2960325.

[4] Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013. ISBN: 9781782161042.

[5] Pierre A. Akiki et al. "EUD-MARS: End-user development of model-driven adaptive robotics software systems." In: *Science of Computer Programming* 200 (2020), p. 102534. ISSN: 0167-6423. DOI: 10.1016/j.scico.2020.102534.

[6] Bushra Alkadhi et al. "Labenah: An Arabic Block-Based Interactive Programming Environment for Children. The Journey of Learning and Playing." In: *HCI International 2019 - Posters*. Springer, 2019, pp. 179–187. ISBN: 978-3-030-23525-3.

[7] Abrar Almjally, Kate Howland, and Judith Good. "Comparing TUIs and GUIs for Primary School Programming." In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: ACM, 2020, pp. 521–527. ISBN: 9781450367936. DOI: 10.1145/3328778.3366851.

[8] I. Almusaly, R. Metoyer, and C. Jensen. "Evaluation of A Visual Programming Keyboard on Touchscreen Devices." In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 57–64. DOI: 10.1109/VLHCC.2018.8506557.

[9] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. "Comparison of Block-Based and Hybrid-Based Environments in Transferring Programming Skills to Text-Based Environments." In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. CASCON '19. Toronto, Ontario, Canada: IBM Corp., 2019, pp. 100–109.

[10] Márcia Alves, Armando Sousa, and Ângela Cardoso. "Web Based Robotic Simulator for Tactode Tangible Block Programming System." In: *Robot 2019: Fourth Iberian Robotics Conference*. Springer, 2020, pp. 490–501. ISBN: 978-3-030-35990-4.

[11] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd. Cambridge University Press, 2003. ISBN: 052182060X.

[12] Ian Arawjo et al. "Teaching Programming with Gamified Semantics." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: ACM, 2017, pp. 4911–4923. ISBN: 978-1-4503-4655-9. DOI: 10.1145/3025453.3025711.

[13]   Adiel Ashrov et al. "A use-case for behavioral programming: An architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios." In: *Science of Computer Programming* 98 (2015), pp. 268–292. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.01.017.

[14]   Umit Aslan et al. "Phenomenological Programming: A Novel Approach to Designing Domain Specific Programming Environments for Science Learning." In: *Proceedings of the Interaction Design and Children Conference*. IDC '20. London, United Kingdom: ACM, 2020, pp. 299–310. ISBN: 9781450379816. DOI: 10.1145/3392063.3394428.

[15]   Egidio Astesiano. "Inductive and Operational Semantics." In: *IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts*. Springer, 1991, pp. 51–136.

[16]   Carlos Pereira Atencio. *Ardublockly - Visual Programming for Arduino*. 2021. URL: https://ardublockly.embeddedlog.com/index.html (visited on 02/22/2021).

[17]   Jack Atherton and Paulo Blikstein. *Define blocks*. 2017. URL: https://ccrma.stanford.edu/~lja/sonification/.

[18]   Jack Atherton and Paulo Blikstein. "Sonification Blocks: A Block-Based Programming Environment For Embodied Data Sonification." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Stanford, California, USA: ACM, 2017, pp. 733–736. ISBN: 9781450349215. DOI: 10.1145/3078072.3091992.

[19]   Isabelle Attali et al. "SmartTools: A Generator of Interactive Environments Tools." In: *Electronic Notes in Theoretical Computer Science* 44.2 (2001). LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001), pp. 225–231. ISSN: 1571-0661.

[20]   Casper Bach Poulsen and Peter D. Mosses. "Generating Specialized Interpreters for Modular Structural Operational Semantics." In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 2014, pp. 220–236. DOI: 10.1007/978-3-319-14125-1_13.

[21]   P. Bachiller-Burgos et al. "LearnBlock: A Robot-Agnostic Educational Programming Tool." In: *IEEE Access* 8 (2020), pp. 30012–30026. DOI: 10.1109/ACCESS.2020.2972410.

[22]   C. Bain et al. "Position: Building Blocks for Agent-based Modeling Can Scaffold Computational Thinking Engagement in STEM Classrooms." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 1–4. DOI: 10.1109/BB48857.2019.8941204.

[23]   M. Bajzek et al. "MUzECS: Embedded blocks for exploring computer science." In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015, pp. 127–132. DOI: 10.1109/BLOCKS.2015.7369021.

[24]   Thomas Ball et al. "Microsoft MakeCode: Embedded Programming for Education, in Blocks and TypeScript." In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. Athens, Greece: ACM, 2019, pp. 7–12. ISBN: 9781450369893. DOI: 10.1145/3358711.3361630.

[25]   Lorena A. Barba. *Teaching and Learning with Jupyter*. https://bit.ly/3zju8ww. [Online, accessed 02 January 2021]. 2019.

[26]   Jur Bartels. "Bridging the worlds of textual and projectional language workbenches." MA thesis. Eindhoven University of Technology, 2020.

[27] Jur Bartels and Mauricio Verano Merino. *Rascal2MPS*. https://github.com/cwi-swat/rascal-mps. [Online, accessed 29 July 2021]. 2019.

[28] Bas Basten et al. "Modular language implementation in Rascal – experience report." In: *Science of Computer Programming* 114 (2015), pp. 7–19. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.11.003.

[29] David Bau. "Droplet, a blocks-based editor for text code." In: *Journal of Computing Sciences in Colleges* 30.6 (2015), pp. 138–144. ISSN: 1937-4771.

[30] David Bau et al. "Learnable Programming: Blocks and Beyond." In: *Commun. ACM* 60.6 (2017), pp. 72–80. ISSN: 0001-0782. DOI: 10.1145/3015455.

[31] David Bau et al. "Learnable Programming: Blocks and Beyond." In: *Commun. ACM* 60.6 (May 2017), pp. 72–80. ISSN: 0001-0782. DOI: 10.1145/3015455.

[32] Tom Beckmann and Mauricio Verano Merino. *maveme/skogi: SKogi 0.1.0*. Version 0.1.0. Sept. 2021. DOI: 10.5281/zenodo.5534113.

[33] Andrew Begel. "LogoBlocks: A Graphical Programming Language for Interacting with the World." MA thesis. Massachusetts Institute of Technology, Media Laboratory., 1996.

[34] Francisco Bellas et al. "The Robobo Project: Bringing Educational Robotics Closer to Real-World Applications." In: *Robotics in Education*. Springer, 2018, pp. 226–237. ISBN: 978-3-319-62875-2.

[35] Francisco Bellas et al. "STEAM Approach to Autonomous Robotics Curriculum for High School Using the Robobo Robot." In: *Robotics in Education*. Springer, 2020, pp. 77–89. ISBN: 978-3-030-26945-6.

[36] Luciana Benotti, Marcos J. Gómez, and Cecilia Martínez. "UNC++Duino: A Kit for Learning to Program Robots in Python and C++ Starting from Blocks." In: *Robotics in Education*. Springer, 2017, pp. 181–192. ISBN: 978-3-319-42975-5.

[37] Laura Benton et al. "Designing for learning mathematics through programming: A case study of pupils engaging with place value." In: *International Journal of Child-Computer Interaction* 16 (2018), pp. 68–76. ISSN: 2212-8689. DOI: 10.1016/j.ijcci.2017.12.004.

[38] Sara Beschi, Daniela Fogli, and Fabio Tampalini. "CAPIRCI: A Multi-modal System for Collaborative Robot Programming." In: *End-User Development*. Springer, 2019, pp. 51–66. ISBN: 978-3-030-24781-2.

[39] M. Beth Kery and B. A. Myers. "Exploring exploratory programming." In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446.

[40] L. Thomas van Binsbergen. *eFLINT implementation on GitLab*. https://bit.ly/3mQpO2G. [Online, accessed 12 October 2020]. 2020.

[41] L. Thomas van Binsbergen and Mauricio Verano Merino. *MiniJava Rascal*. https://github.com/cwi-swat/rascal-minijava. [Online, accessed 29 July 2021]. 2019.

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

[43] L. Thomas van Binsbergen, Mauricio Verano Merino, and Tom Beckmann. *MiniJava Syntax*. https://bit.ly/3FRs5SK. [Online, accessed 03 January 2022]. 2020.

[44] L. Thomas van Binsbergen et al. "A Principled Approach to REPL Interpreters." In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. ACM, 2020, pp. 84–100. ISBN: 9781450381789. DOI: 10.1145/3426428.3426917.

[45] L. Thomas van Binsbergen et al. "eFLINT: A Domain-Specific Language for Executable Norm Specifications." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2020. ACM, 2020. DOI: 10.1145/3425898.3426958.

[46] A. F. Blackwell et al. "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology." In: *Cognitive Technology: Instruments of Mind*. Springer, 2001, pp. 325–341. ISBN: 978-3-540-44617-0.

[47] A. F. Blackwell et al. "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology." In: *Cognitive Technology: Instruments of Mind*. Ed. by Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn. Berlin, Heidelberg: Springer, 2001, pp. 325–341. ISBN: 978-3-540-44617-0.

[48] J. Blanchard, C. Gardner-McCune, and L. Anthony. "Amphibian: Dual-Modality Representation in Integrated Development Environments." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 83–85. DOI: 10.1109/BB48857.2019.8941213.

[49] Geoff Boeing and Dani Arribas-Bel. *GIS and Computational Notebooks*. 2021. arXiv: 2101.00351 [cs.CY].

[50] Rodrigo Bonifacio. *Rascal-Java8*. https://bit.ly/3mN9JLu. [Online, accessed 15 July 2021]. 2008.

[51] D. Bonino et al. "Block-based realtime big-data processing for smart cities." In: *2016 IEEE International Smart Cities Conference (ISC2)*. 2016, pp. 1–6. DOI: 10.1109/ISC2.2016.7580768.

[52] Erwan Bousse et al. "Omniscient debugging for executable DSLs." In: *Journal of Systems and Software* 137 (2018), pp. 261–288. DOI: 10.1016/j.jss.2017.11.025.

[53] Mark G. J. van den Brand et al. "The ASF+SDF Meta-Environment: A Component-Based Language Development Environment." In: *Proceedings of the 10th International Conference on Compiler Construction*. CC '01. Springer, 2001, pp. 365–370. ISBN: 354041861X.

[54] Mark G.J. van den Brand et al. "The ASF+SDF Meta-Environment: A Component-Based Language Development Environment." In: *Electronic Notes in Theoretical Computer Science* 44.2 (2001). LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001), pp. 3–8. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80917-4.

[55] Mark van den Brand and Eelco Visser. "Generation of formatters for context-free languages." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.1 (1996), pp. 1–41.

[56] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation." In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

[57] Benedikt Breuch and Martin Fislake. "First Steps in Teaching Robotics with Drones." In: *Robotics in Education*. Springer, 2020, pp. 138–144. ISBN: 978-3-030-26945-6.

[58] Neil C.C. Brown et al. "Panel: Future Directions of Block-based Programming." In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: ACM, 2016, pp. 315–316. ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844661.

[59] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. "Domain-specific Software Engineering." In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 65–68. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882376. URL: http://doi.acm.org/10.1145/1882362.1882376.

[60] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003. ISBN: 0131425420.

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

[62] Fabien Campagne and Fabien Campagne. *The MPS Language Workbench, Vol. 1*. 1st. CreateSpace Independent Publishing Platform, 2014. ISBN: 1497378656.

[63] André Campos et al. "piBook: Introducing Computational Thinking to Diversified Audiences." In: *Computers Supported Education*. Springer, 2018, pp. 179–195. ISBN: 978-3-319-94640-5.

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

[65] M. Cápay and N. Klimová. "Engage Your Students via Physical Computing!" In: *2019 IEEE Global Engineering Education Conference (EDUCON)*. 2019, pp. 1216–1223. DOI: 10.1109/EDUCON.2019.8725101.

[66] Logan B. Caraco et al. "Making the Blockly Library Accessible via Touchscreen." In: *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '19. Pittsburgh, PA, USA: ACM, 2019, pp. 648–650. ISBN: 9781450366762. DOI: 10.1145/3308561.3354589.

[67] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton. "IMP: A Meta-Tooling Platform for Creating Language-Specific Ides in Eclipse." In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 485–488. ISBN: 9781595938824. DOI: 10.1145/1321631.1321715.

[68] Philippe Charles et al. "Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse." In: 44.10 (2009), pp. 191–206. ISSN: 0362-1340. DOI: 10.1145/1639949.1640104.

[69] Souti Chattopadhyay et al. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: ACM, 2020, pp. 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376729.

[70] Avraam Chatzopoulos et al. "Innovative Robot for Educational Robotics and STEM." In: *Intelligent Tutoring Systems*. Springer, 2020, pp. 95–104. ISBN: 978-3-030-49663-0.

[71] Jacob Cohen. "A Coefficient of Agreement for Nominal Scales." In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46. DOI: 10.1177/001316446002000104.

[72] Joshua Cook. "Interactive Programming." In: *Docker for Data Science: Building Scalable and Extensible Data Infrastructure Around the Jupyter Notebook Server*. Apress, 2017, pp. 49–70. ISBN: 978-1-4842-3012-1. DOI: 10.1007/978-1-4842-3012-1_3.

[73] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello B. "My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor." In: 11553 (2019), pp. 18–33. DOI: 10.1007/978-3-030-24781-2.

[74] Enrique Coronado et al. "Visual Programming Environments for End-User Development of intelligent and social robots, a systematic review." In: *Journal of Computer Languages* 58 (2020), p. 100970. ISSN: 2590-1184. DOI: 10.1016/j.cola.2020.100970.

[75] A. Cory Bart et al. "Design and Evaluation of a Block-based Environment with a Data Science Context." In: *IEEE Transactions on Emerging Topics in Computing* (2017), pp. 1–1. DOI: 10.1109/TETC.2017.2729585.

[76] Chris S. Crawford, Christina Gardner-McCune, and Juan E. Gilbert. "Brain-Computer Interface for Novice Programmers." In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: ACM, 2018, pp. 32–37. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159603.

[77] Chris S. Crawford and Juan E. Gilbert. "Brains and Blocks: Introducing Novice Programmers to Brain-Computer Interface Application Development." In: *ACM Trans. Comput. Educ.* 19.4 (2019), 39:1–39:27. ISSN: 1946-6226. DOI: 10.1145/3335815.

[78] CWI-SWAT. *SweeterJS*. https://bit.ly/3FQgHGY. [Online, accessed 12 July 2021]. 2019.

[79] CWI-SWAT. *Syntax Definition*. 2020. URL: https://bit.ly/3HzocTb (visited on 03/26/2020).

[80] CWI-SWAT. *Hack your javascript*. 2019. URL: https://github.com/cwi-swat/hack-your-javascript (visited on 12/22/2019).

[81] Dagre. *DagreJS*. URL: https://github.com/dagrejs (visited on 12/22/2019).

[82] Al Danial. *Cloc*. 2006. URL: https://github.com/AlDanial/cloc.

[83] Meenakshi Das et al. "Accessible Computer Science for K-12 Students with Hearing Impairments." In: *Universal Access in Human-Computer Interaction. Applications and Practice*. Springer, 2020, pp. 173–183. ISBN: 978-3-030-49108-6.

[84] Sayamindu Dasgupta and Benjamin Mako Hill. "Scratch Community Blocks: Supporting Children As Data Scientists." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: ACM, 2017, pp. 3620–3631. ISBN: 978-1-4503-4655-9. DOI: 10.1145/3025453.3025847.

[85] Alan Davies et al. "Using interactive digital notebooks for bioscience and informatics education." In: *PLOS Computational Biology* 16.11 (Nov. 2020), pp. 1–19. DOI: 10.1371/journal.pcbi.1008326.

[86] Merijn De Jonge. "Pretty-printing for software reengineering." In: *International Conference on Software Maintenance, 2002. Proceedings.* IEEE. 2002, pp. 550–559.

[87] A. Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. AMAST series in computing. World Scientific, 1996. ISBN: 9789810227326. URL: https://books.google.nl/books?id=GT1qDQAAQBAJ.

[88] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography." In: *SIGPLAN Not.* 35.6 (2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035.

[89] Peter Deutsch. *PDP-1 LISP*. Tech. rep. https://bit.ly/3G4CUkR. MIT Research Laboratory for Electronics, 1964.

[90] James Devine et al. "MakeCode and CODAL: Intuitive and efficient embedded systems programming for education." In: *Journal of Systems Architecture* 98 (2019), pp. 468–483. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2019.05.005.

[91] Shruti Dhariwal. "BlockArt: Visualizing the 'Hundred Languages' of Code in Children's Creations." In: *Proceedings of the 2019 on Creativity and Cognition*. C&C '19. San Diego, CA, USA: ACM, 2019, pp. 633–639. ISBN: 978-1-4503-5917-7. DOI: 10.1145/3325480.3326585.

[92] Raffaele Di Fuccio et al. "Digital and Multisensory Storytelling: Narration with Smell, Taste and Touch." In: *Adaptive and Adaptable Learning*. Springer, 2016, pp. 509–512. ISBN: 978-3-319-45153-4. DOI: 10.1007/978-3-319-45153-4_51.

[93] Sergey Dmitriev. *Language Oriented Programming: The Next Programming Paradigm*. Tech. rep. Jetbrains, 2004, p. 14. URL: https://bit.ly/3ET9Cnv.

[94] Yihuan Dong et al. "Defining Tinkering Behavior in Open-ended Block-based Programming Assignments." In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: ACM, 2019, pp. 1204–1210. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287437.

[95] Véronique Donzeau-Gouge et al. "Programming environments based on structured editors : the Mentor experience." In: *Interact Program Environ* (1984).

[96] Hilary Dwyer et al. "Fourth Grade Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances." In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. Omaha, Nebraska, USA: ACM, 2015, pp. 111–119. ISBN: 978-1-4503-3630-7. DOI: 10.1145/2787622.2787729.

[97] Alexander Eiselmayer. "Supporting the Design and Analysis of HCI Experiments." In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: ACM, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3375038.

[98] Söderberg Emma and Hedin Görel. "Building Semantic Editors Using JastAdd: Tool Demonstration." In: LDTA '11 (2011). DOI: 10.1145/1988783.1988794.

[99] S. Erdweg et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future." In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.08.007.

[100] Sebastian Erdweg et al. "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge." In: *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*. 2013, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.

[101] Sarah Esper, Stephen R. Foster, and William G. Griswold. "On the Nature of Fires and How to Spark Them When You're Not There." In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: ACM, 2013, pp. 305–310. ISBN: 9781450318686. DOI: 10.1145/2445196.2445290.

[102] Moritz Eysholdt and Heiko Behrens. "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*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 307–309. ISBN: 9781450302401. DOI: 10.1145/1869542.1869625.

[103] P. G. Feijó-García et al. "Design and evaluation of a scaffolded block-based learning environment for hierarchical data structures." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 145–149. DOI: 10.1109/VLHCC.2019.8818759.

[104] A. Feng and W. Feng. "Parallel Programming with Pictures in a Snap!" In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 950–957. DOI: 10.1109/IPDPSW.2016.194.

[105] A. Feng, E. Tilevich, and W. Feng. "Block-based programming abstractions for explicit parallel computing." In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015, pp. 71–75. DOI: 10.1109/BLOCKS.2015.7369006.

[106] Annette Feng, Mark Gardner, and Wu-chun Feng. "Parallel programming with pictures is a Snap!" In: *Journal of Parallel and Distributed Computing* 105 (2017), pp. 150–162. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2017.01.018.

[107] Cassia Fernandez. "Combining Computational and Science Practices in K-12 Education." In: *Proceedings of the 2020 ACM Interaction Design and Children Conference: Extended Abstracts*. IDC '20. London, United Kingdom: ACM, 2020, pp. 25–29. ISBN: 9781450380201. DOI: 10.1145/3397617.3398033.

[108] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation." In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 111–122. ISBN: 158113729X. DOI: 10.1145/964001.964011.

[109] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation." In: *SIGPLAN Not.* 39.1 (Jan. 2004), pp. 111–122. ISSN: 0362-1340. DOI: 10.1145/982962.964011.

[110] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* https://bit.ly/32YuhJT. [Online, accessed 11 August 2021]. 2015.

[111] Christoph Frädrich et al. "Common Bugs in Scratch Programs." In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '20. Trondheim, Norway: ACM, 2020, pp. 89–95. ISBN: 9781450368742. DOI: 10.1145/3341525.3387389.

[112] Diana Franklin et al. "Initialization in Scratch: Seeking Knowledge Transfer." In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: ACM, 2016, pp. 217–222. ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844569.

[113]   Maurizio Gabbrielli and Simone Martini. "How to Describe a Programming Language." In: *Programming Languages: Principles and Paradigms*. Springer, 2010, pp. 27–55. ISBN: 978-1-84882-914-5. DOI: 10.1007/978-1-84882-914-5_2.

[114]   D.J. Gilmore and T.R.G. Green. "Comprehension and recall of miniature programs." In: *International Journal of Man-Machine Studies* 21.1 (1984), pp. 31–48. ISSN: 0020-7373.

[115]   Terrell Glenn et al. "StoryMakAR: Bringing Stories to Life With An Augmented Reality & Physical Prototyping Toolkit for Youth." In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: ACM, 2020, pp. 1–14. ISBN: 9781450367080. DOI: 10.1145/3313831.3376790.

[116]   J. A. Goguen et al. "Initial Algebra Semantics and Continuous Algebras." In: *Journal of the ACM* 24.1 (1977), pp. 68–95. ISSN: 0004-5411. DOI: 10.1145/321992.321997.

[117]   Gilad Gome et al. "OpenLH: Open Liquid-Handling System for Creative Experimentation with Biology." In: *Proceedings of the Thirteenth International Conference on Tangible, Embedded, and Embodied Interaction*. TEI '19. Tempe, Arizona, USA: ACM, 2019, pp. 55–64. ISBN: 9781450361965. DOI: 10.1145/3294109.3295619.

[118]   Judith Good and Kate Howland. "Programming language, natural language? Supporting the diverse computational activities of novice programmers." In: *Journal of Visual Languages & Computing* 39 (2017), pp. 78–92. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2016.10.008.

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

[120]   Google. *Define input blocks*. 2020. URL: https://bit.ly/3pLDRbJ.

[121]   Google. *Define output blocks*. 2020. URL: https://bit.ly/3FS6ny5.

[122]   Google. *Custom Blocks: Block Paradigms*. 2021. URL: https://bit.ly/370143q (visited on 03/20/2021).

[123]   Google. *Google Charts*. URL: https://developers.google.com/chart/ (visited on 12/22/2019).

[124]   Jamie Gorson et al. "TunePad: Computational Thinking Through Sound Composition." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Stanford, California, USA: ACM, 2017, pp. 484–489. ISBN: 978-1-4503-4921-5. DOI: 10.1145/3078072.3084313.

[125]   Maria Gouseti, Chiel Peters, and Tijs van der Storm. "Extensible Language Implementation with Object Algebras (Short Paper)." In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE'14)*. Västerås, Sweden, 2014, pp. 25–28. DOI: 10.1145/2658761.2658765.

[126]   Thomas R. G. Green and Marian Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." In: *J. Vis. Lang. Comput.* 7.2 (1996), pp. 131–174. DOI: 10.1006/jvlc.1996.0009.

[127]   Thomas Green and Alan Blackwell. "Cognitive dimensions of information artefacts: a tutorial." In: *BCS HCI Conference*. Vol. 98. 1998, pp. 1–75.

[128]   Andreas Grillenberger and Ralf Romeike. "Real-Time Data Analyses in Secondary Schools Using a Block-Based Programming Language." In: *Informatics in Schools: Focus on Learning Programming*. Springer, 2017, pp. 207–218. ISBN: 978-3-319-71483-7.

[129]   Marianthi Grizioti and Chronis Kynigos. "Game Modding for Computational Thinking: An Integrated Design Approach." In: *Proceedings of the 17th ACM Conference on Interaction Design and Children*. IDC '18. Trondheim, Norway: ACM, 2018, pp. 687–692. ISBN: 978-1-4503-5152-2. DOI: 10.1145/3202185.3210800.

[130]   Philip J. Guo and Margo Seltzer. "BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure." In: *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*. TaPP'12. Boston, MA: USENIX Association, 2012, p. 7.

[131]   Marijn Haverbeke. *CodeMirror*. 2007–2018. URL: http://codemirror.net/.

[132]   Brian Hayes. "Thoughts on Mathematica." In: *Pixel* 1.January/February (1990), pp. 28–34.

[133]   Andrew Head et al. "Managing Messes in Computational Notebooks." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19* (2019), pp. 1–12. DOI: 10.1145/3290605.3300500.

[134]   Jan Heering et al. "The syntax definition formalism SDF - reference manual." In: *ACM SIGPLAN Notices* 24.11 (1989), pp. 43–75. DOI: 10.1145/71605.71607.

[135]   Brian Hempel, Justin Lubin, and Ravi Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG." In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: ACM, 2019, pp. 281–292. ISBN: 9781450368162. DOI: 10.1145/3332165.3347925.

[136]   F. Hermans and E. Aivaloglou. "Do code smells hamper novice programming? A controlled experiment on Scratch programs." In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503706.

[137]   Bryan Hernandez-Cuevas et al. "Changing Minds: Exploring Brain-Computer Interface Experiences with High School Students." In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: ACM, 2020, pp. 1–10. ISBN: 9781450368193. DOI: 10.1145/3334480.3382981.

[138]   Charlotte Hill et al. "Floors and Flexibility: Designing a Programming Environment for 4Th-6th Grade Classrooms." In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 546–551. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677275.

[139]   R. Holwerda and F. Hermans. "A Usability Analysis of Blocks-based Programming Editors using Cognitive Dimensions." In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 217–225. DOI: 10.1109/VLHCC.2018.8506483.

[140]   Sheng-Yi Hsu et al. "Shelves: A User-Defined Block Management Tool for Visual Programming Languages." In: *Human-Computer Interaction – INTERACT 2017*. Springer, 2017, pp. 335–344. ISBN: 978-3-319-67687-6.

[141]   IFTTT. *IFTTT*. 2021. URL: https://ifttt.com (visited on 03/01/2021).

[142] MathWorks Inc. *MATLAB Live Editor*. 2019. URL: https://bit.ly/3sNENyp (visited on 07/30/2019).

[143] Dan Ingalls et al. "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*. OOPSLA '97. Atlanta, Georgia, USA: ACM, 1997, pp. 318–326. ISBN: 0897919084. DOI: 10.1145/263698.263754.

[144] Pablo Inostroza, Tijs van der Storm, and Sebastian Erdweg. "Tracing Program Transformations with String Origins." In: *Theory and Practice of Model Transformations*. Springer, 2014, pp. 154–169. ISBN: 978-3-319-08789-4.

[145] A. Islam et al. "EduBot: An Educational Robot for Underprivileged Children." In: *2019 International Conference on Automation, Computational and Technology Management (ICACTM)*. 2019, pp. 232–236. DOI: 10.1109/ICACTM.2019.8776756.

[146] K. Ito. "Work in Progress: Block Pictogramming A Block-based Programming Learning Environment through Pictogram Content Creation." In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. 2020, pp. 1669–1673. DOI: 10.1109/EDUCON45650.2020.9125386.

[147] Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, and Jesús Garcıa Molina. "Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization." In: *Workshop on Model-Driven Software Evolution*. 2008, pp. 1–8.

[148] Heering Jan and Klint Paul. "Semantics of Programming Languages: A Tool-oriented Approach." In: *SIGPLAN Not.* 35.3 (2000), pp. 39–48. ISSN: 0362-1340. DOI: 10.1145/351159.351173.

[149] B. Jansen and F. Hermans. "XLBlocks: a Block-based Formula Editor for Spreadsheet Formulas." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 55–63. DOI: 10.1109/VLHCC.2019.8818748.

[150] S. Jatzlau, S. Seegerer, and R. Romeike. "The Five Million Piece Puzzle: Finding Answers in 500,000 Snap!-Projects." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 73–77. DOI: 10.1109/BB48857.2019.8941206.

[151] S. Jatzlau et al. "It's not Magic After All – Machine Learning in Snap! using Reinforcement Learning." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 37–41. DOI: 10.1109/BB48857.2019.8941208.

[152] Pierre Jeanjean, Benoit Combemale, and Olivier Barais. "From DSL Specification to Interactive Computer Programming Environment." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE'19)*. Athens, Greece, 2019, pp. 167–178. ISBN: 9781450369817. DOI: 10.1145/3357766.3359540.

[153] Jeremiah W. Johnson and Karen H. Jin. "Jupyter Notebooks in Education." In: *J. Comput. Sci. Coll.* 35.8 (Apr. 2020), pp. 268–269. ISSN: 1937-4771.

[154] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Tech. rep. 1979.

[155] Merijn de Jonge, Eelco Visser, and Joost Visser. "XT: a bundle of program transformation tools." In: *Electron. Notes Theor. Comput. Sci.* 44.2 (2001), pp. 79–86. DOI: 10.1016/S1571-0661(04)80921-6.

[156] I. Jung et al. "Interactive learning environment for practical programming language based on web service." In: *2016 15th International Conference on Information Technology Based Higher Education and Training (ITHET)*. 2016, pp. 1–7. DOI: 10.1109/ITHET.2016.7760705.

[157] Jupyter.org. *The wire protocol*. 2015. URL: https://bit.ly/3JD49oI (visited on 07/24/2017).

[158] Jupyter.org. *Jupyter kernels*. 2019. URL: https://bit.ly/3EPT9jY (visited on 07/30/2019).

[159] Jupyter.org. *Making kernels for Jupyter*. 2019. URL: https://bit.ly/34g8n5s (visited on 07/26/2019).

[160] Jupyter.org. *Making simple Python wrapper kernels*. 2019. URL: https://bit.ly/3zi5H2u (visited on 09/25/2019).

[161] Jupyter.org. *The Jupyter Notebook*. https://jupyter.org. [Online, accessed 06 August 2021]. 2019.

[162] Gilles Kahn. "Natural Semantics." In: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1987, pp. 22–39. DOI: 10.1007/BFb0039592.

[163] Kyo C. Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Software Engineering Institute, 1990.

[164] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, 1990. URL: https://bit.ly/31mK6JS.

[165] Kantinee Katchapakirin and Chutiporn Anutariya. "An Architectural Design of ScratchThAI: A Conversational Agent for Computational Thinking Development Using Scratch." In: *Proceedings of the 10th International Conference on Advances in Information Technology*. IAIT 2018. Bangkok, Thailand: ACM, 2018, 7:1–7:7. ISBN: 978-1-4503-6568-0. DOI: 10.1145/3291280.3291787.

[166] Lennart C.L. Kats and Eelco Visser. "The Spoofax 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*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 444–463. ISBN: 9781450302036. DOI: 10.1145/1869459.1869497.

[167] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The IDE Portability Problem and Its Solution in Monto." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 152–162. ISBN: 9781450344470. DOI: 10.1145/2997364.2997368.

[168] Caitlin Kelleher and Randy Pausch. "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers." In: *ACM Comput. Surv.* 37.2 (2005), pp. 83–137. ISSN: 0360-0300. DOI: 10.1145/1089733.1089734.

[169] Annie Kelly et al. "ARcadia: A Rapid Prototyping Platform for Real-time Tangible Interfaces." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 409:1–409:8. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173983.

[170] Annie Kelly et al. "BlockyTalky: New programmable tools to enable students' learning networks." In: *International Journal of Child-Computer Interaction* 18 (2018), pp. 8–18. ISSN: 2212-8689. DOI: 10.1016/j.ijcci.2018.03.004.

[171] S. Kelly, K. Lyytinen, and M. Rossi. "MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment." In: *CAiSE*. 1996.

[172] Mary B. Kery et al. "The story in the notebook: Exploratory data science using a literate programming tool." In: *Conference on Human Factors in Computing Systems - Proceedings* 2018-April (2018), pp. 1–11. DOI: 10.1145/3173574.3173748.

[173] Mary Beth Kery, Amber Horvath, and Brad Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: ACM, 2017, pp. 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626.

[174] Mary Beth Kery et al. "mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks." In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (2020). DOI: 10.1145/3379337.3415842.

[175] Mary Beth Kery et al. "The Future of Notebook Programming Is Fluid." In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: ACM, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383085.

[176] Daehoon Kim et al. "Pedagogy of Programming Education for Higher Education Using Block Based Programming Environment." In: *Design, User Experience, and Usability: Designing Interactions*. Springer, 2018, pp. 39–50. ISBN: 978-3-319-91803-7.

[177] Barbara Ann Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007.

[178] Barbara Kitchenham et al. "Systematic literature reviews in software engineering – A systematic literature review." In: *Information and Software Technology* 51.1 (2009), pp. 7–15. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.09.009.

[179] P. Klint, T. v. d. Storm, and J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation." In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2009, pp. 168–177.

[180] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "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*. IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28.

[181] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows." In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, pp. 87–90.

[182] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows." In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 2016, pp. 87–90.

[183] D. E. Knuth. "Literate Programming." In: *The Computer Journal* 27.2 (1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97.

[184] Andrew J. Ko et al. "The state of the art in end-user software engineering." In: *ACM Computing Surveys* 43.3 (2011), pp. 1–44. ISSN: 03600300. DOI: 10.1145/1922649.1922658.

[185] W. Koehler and Y. Jing. "A Novel Block-Based Programming Framework for Non-programmers to Validate PLC Based Machine Tools for Automotive Manufacturing Facilities." In: *2018 11th International Conference on Developments in eSystems Engineering (DeSE)*. 2018, pp. 202–207. DOI: 10.1109/DeSE.2018.00046.

[186] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. "Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming." In: *Proceedings of the Workshop in Primary and Secondary Computing Education*. WiPSCE '15. London, United Kingdom: ACM, 2015, pp. 29–38. ISBN: 978-1-4503-3753-3. DOI: 10.1145/2818314.2818331.

[187] Michael Kölling et al. "Stride in BlueJ – Computing for All in an Educational IDE." In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: ACM, 2019, pp. 63–69. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287462.

[188] Siu-Cheung Kong. "Learning Composite and Prime Numbers Through Developing an App: An Example of Computational Thinking Development Through Primary Mathematics Learning." In: *Computational Thinking Education*. Springer, 2019, pp. 145–166. ISBN: 978-981-13-6528-7. DOI: 10.1007/978-981-13-6528-7_9.

[189] David Koop and Jay Patel. "Dataflow Notebooks: Encoding and Tracking Dependencies of Cells." In: *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*. TaPP'17. Seattle, WA: USENIX Association, 2017, p. 17.

[190] Chonnuttida Koracharkornradt. "Tuk Tuk: A Block-Based Programming Game." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Stanford, California, USA: ACM, 2017, pp. 725–728. ISBN: 978-1-4503-4921-5. DOI: 10.1145/3078072.3091990.

[191] Varsha Koushik, Darren Guinness, and Shaun K. Kane. "StoryBlocks: A Tangible Programming Game To Create Accessible Audio Stories." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: ACM, 2019, 492:1–492:12. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300722.

[192] Charles W. Krueger. "Software Reuse." In: *ACM Comput. Surv.* 24.2 (1992), pp. 131–183. ISSN: 0360-0300. DOI: 10.1145/130844.130856.

[193] Josiah Krutz et al. "Stepwise Refinement in Block-Based Programming." In: *J. Comput. Sci. Coll.* 35.5 (2019), pp. 91–100. ISSN: 1937-4771.

[194] Azusa Kurihara et al. "A Programming Environment for Visual Block-Based Domain-Specific Languages." In: *Procedia Computer Science* 62 (2015). Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15), pp. 287–296. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.08.452.

[195] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. "Technological spaces: An initial appraisal." In: *CoopIS, DOA'2002 Federated Conferences, Industrial track*. 2002.

[196] C. Kyfonidis, N. Moumoutzis, and S. Christodoulakis. "Block-C: A block-based programming teaching tool to facilitate introductory C programming courses." In: *2017 IEEE Global Engineering Education Conference (EDUCON)*. 2017, pp. 570–579. DOI: 10.1109/EDUCON.2017.7942903.

[197] Ralf Lämmel. "The Notion of a Software Language." In: *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018, pp. 1–49. ISBN: 978-3-319-90800-7. DOI: 10.1007/978-3-319-90800-7_1.

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

[199] J. Richard Landis and Gary G. Koch. "The Measurement of Observer Agreement for Categorical Data." In: *Biometrics* 33.1 (1977), pp. 159–174. ISSN: 0006341X, 15410420.

[200] Microsoft. *Language Server Protocol*. 2018. URL: https://bit.ly/3EPnzDh.

[201] S. Lau et al. "The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry." In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

[202] Ž. Leber, M. Črepinek, and T. Kosar. "Simultaneous multiple representation editing environment for primary school education." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 175–179. DOI: 10.1109/VLHCC.2019.8818927.

[203] Akos Ledeczi et al. "The Generic Modeling Environment." In: *Workshop on Intelligent Signal Processing*. 2001.

[204] Ákos Lédeczi et al. "Teaching Cybersecurity with Networked Robots." In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: ACM, 2019, pp. 885–891. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287450.

[205] Beng Yong Lee et al. "The Effectiveness of Using MBot to Increase the Interest and Basic Knowledge in Programming and Robotic among Children of Age 13." In: *Proceedings of the 2020 The 6th International Conference on E-Business and Applications*. ICEBA 2020. Kuala Lumpur, Malaysia: ACM, 2020, pp. 105–110. ISBN: 9781450377355. DOI: 10.1145/3387263.3387275.

[206] Lap-Kei Lee et al. "Learning Computational Thinking Through Gamification and Collaborative Learning." In: *Blended Learning: Educational Innovation for Personalized Learning*. Springer, 2019, pp. 339–349. ISBN: 978-3-030-21562-0.

[207] Sorin Lerner, Stephen R. Foster, and William G. Griswold. "Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors." In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: ACM, 2015, pp. 3063–3072. ISBN: 978-1-4503-3145-6. DOI: 10.1145/2702123.2702302.

[208] Bil Lewis. "Debugging Backwards in Time." In: *Computing Research Repository* cs.SE/0310016 (2003). URL: http://arxiv.org/abs/cs/0310016.

[209] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters." In: *22nd Symposium on Principles of Programming Languages*. ACM, 1995, pp. 333–343. DOI: 10.1145/199448.199528.

[210] Henry Lieberman et al. "End-User Development: An Emerging Paradigm." In: *End User Development*. Springer, 2006, pp. 1–8. ISBN: 978-1-4020-5386-3. DOI: `10.1007/1-4020-5386-X\_1`.

[211] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical object-oriented back-in-time debugging." In: *European Conference on Object-Oriented Programming*. Springer. 2008, pp. 592–615.

[212] N. Lytle et al. "Resource Rush: Towards An Open-Ended Programming Game." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 91–93. DOI: `10.1109/BB48857.2019.8941215`.

[213] Jason Madar. "c3d.io: Enabling STEAM (Science, Technology, Engineering, Arts, Mathematics) Education with Virtual Reality." In: *Intelligent Computing*. Springer, 2019, pp. 1380–1386. ISBN: 978-3-030-01177-2.

[214] Simon Marlow. *Haskell 2010 Language Report*. `https://bit.ly/3EOvikL`. [Online, accessed 12 October 2020]. 2010.

[215] Samiha Marwan, Joseph Jay Williams, and Thomas Price. "An Evaluation of the Impact of Automated Programming Hints on Performance and Learning." In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER '19. Toronto ON, Canada: ACM, 2019, pp. 61–70. ISBN: 978-1-4503-6185-9. DOI: `10.1145/3291279.3339420`.

[216] J. McCarthy et al. *Lisp I programmer's manual*. `http://history.siam.org/sup/Fox_1960_LISP.pdf`. [Online, accessed 12 October 2020]. Computation Center and Research Laboratory of Electronics (MIT). 1960.

[217] Monica M. McGill and Adrienne Decker. "Construction of a Taxonomy for Tools, Languages, and Environments across Computing Education." In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. Virtual Event, New Zealand: ACM, 2020, pp. 124–135. ISBN: 9781450370929. DOI: `10.1145/3372782.3406258`.

[218] Amelia McNamara. "Dynamic Documents with R and knitr." In: *Journal of Statistical Software, Book Reviews* 56.2 (2014), pp. 1–4. ISSN: 1548-7660. DOI: `10.18637/jss.v056.b02`.

[219] A. Mehul et al. "Position: A Novice Oriented Dual-Modality Programming Tool for Brain-Computer Interfaces Application Development." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 27–30. DOI: `10.1109/BB48857.2019.8941226`.

[220] Josh G. M. Mengerink et al. "Exploring DSL Evolutionary Patterns in Practice - A Study of DSL Evolution in a Large-scale Industrial DSL Repository." In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC. SciTePress, 2018, pp. 446–453. ISBN: 978-989-758-283-7. DOI: `10.5220/0006605804460453`.

[221] Mauricio Verano Merino. `https://github.com/maveme/PDFMiner`. [Online, accessed 29 July 2021]. 2019.

[222] Mauricio Verano Merino. `https://github.com/cwi-swat/halide-syntax`. [Online, accessed 29 July 2021]. 2019.

[223] Mauricio Verano Merino. *Execution Graph*. `https://bit.ly/3EKtXvj`. [Online, accessed 29 July 2021]. 2019.

[224]   Mauricio Verano Merino. *maveme/bacata-demos: First release*. Version 1.0.0. Feb. 2020. DOI: `10.5281/zenodo.3636103`.

[225]   Mauricio Verano Merino. *Rascal - Sonification Blocks*. `https://bit.ly/3sRUKn6`. [Online, accessed 12 July 2021]. 2020.

[226]   Mauricio Verano Merino and Tijs van der Storm. "Language Workbench Support for Block-Based DSLs." In: *BLOCKS+ Proceedings* (2018).

[227]   Mauricio Verano Merino and Tijs van der Storm. *cwi-swat/kogi: Kogi 0.1.0*. Version 0.1.1. 2020. DOI: `10.5281/zenodo.4033220`.

[228]   Mauricio Verano Merino and Tijs van der Storm. *cwi-swat/kogi: Kogi 0.1.0*. Version 0.1.1. Sept. 2020. DOI: `10.5281/zenodo.4033220`.

[229]   Mauricio Verano Merino and Jurgen Vinju. *Rascal Notebook*. `https://bit.ly/3zh2Uqt`. [Online, accessed 29 July 2021]. 2019.

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

[231]   Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: a generic notebook generator for DSLs." In: Domain-Specific Language Design and Implementation workshop, DSLDI '17. Vancouver, British Columbia, Canada, 2017.

[232]   Mauricio Verano Merino et al. "Domain-Specific Languages in Practice with Jetbrains MPS." In: 2021. Chap. Projecting Textual Languages. ISBN: 978-3-030-73758-0. DOI: `10.1007/978-3-030-73758-0`.

[233]   Mauricio Verano Merino et al. "Projecting Textual Languages." In: *Domain-Specific Languages in Practice: with JetBrains MPS*. Ed. by Antonio Bucchiarone et al. Cham: Springer, 2021, pp. 197–225. ISBN: 978-3-030-73758-0. DOI: `10.1007/978-3-030-73758-0_7`.

[234]   Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux Journal* 2014.239 (2014). ISSN: 1075-3583.

[235]   Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages." In: *ACM Computing Surveys (CSUR)* 37.4 (2005), pp. 316–344. ISSN: 0360-0300. DOI: `10.1145/1118890.1118892`.

[236]   A. Milliken et al. "Poster: Designing GradeSnap for Block-Based Code." In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–2. DOI: `10.1109/VL/HCC50065.2020.9127284`.

[237]   L. R. Milne and R. E. Ladner. "Position: Accessible Block-Based Programming: Why and How." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 19–22. DOI: `10.1109/BB48857.2019.8941230`.

[238]   Lauren R. Milne and Richard E. Ladner. "Blocks4All: Overcoming Accessibility Barriers to Blocks Programming for Children with Visual Impairments." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 69:1–69:10. ISBN: 978-1-4503-5620-6. DOI: `10.1145/3173574.3173643`.

[239]   Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN: 0262631814.

[240] MIT. *Scratch*. https://scratch.mit.edu. [Online, accessed 29 July 2021]. 2019.

[241] Monika Mladenović, Ivica Boljat, and Žana Žanko. "Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level." In: *Education and Information Technologies* 23.4 (2018), pp. 1483–1500. ISSN: 1573-7608. DOI: 10.1007/s10639-017-9673-3.

[242] Eugenio Moggi. "Notions of Computation and Monads." In: *Information and Computation* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[243] Jens Mönig and Brian Harvey. *Snap!* https://snap.berkeley.edu. [Online, accessed 29 July 2021]. 2019.

[244] Arjan J. Mooij, Jozef Hooman, and Rob Albers. "Gaining Industrial Confidence for the Introduction of Domain-Specific Languages." In: *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. 2013, pp. 662–667. DOI: 10.1109/COMPSACW.2013.83.

[245] L. Moors, A. Luxton-Reilly, and P. Denny. "Transitioning from Block-Based to Text-Based Programming Languages." In: *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 2018, pp. 57–64. DOI: 10.1109/LaTICE.2018.000-5.

[246] Luke Moors and Robert Sheehan. "Aiding the Transition from Novice to Traditional Programming Environments." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Stanford, California, USA: ACM, 2017, pp. 509–514. ISBN: 9781450349215. DOI: 10.1145/3078072.3084317.

[247] Peter D. Mosses. "Modular Structural Operational Semantics." In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 195–228. DOI: 10.1016/j.jlap.2004.03.008.

[248] José Miguel Mota et al. "Augmented reality mobile app development for all." In: *Computers & Electrical Engineering* 65 (2018), pp. 250–260. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2017.08.025.

[249] José Miguel Mota et al. "Learning Analytics in Mobile Applications Based on Multimodal Interaction." In: *Software Data Engineering for Network eLearning Environments: Analytics and Awareness Learning Services*. Springer, 2018, pp. 67–92. ISBN: 978-3-319-68318-8. DOI: 10.1007/978-3-319-68318-8_4.

[250] JetBrains MPS. *Editor Actions*. https://bit.ly/3pN7k5l. [Online, accessed 26 September 2021]. 2021.

[251] M. Müller et al. "Enabling Teenagers to Create and Share Apps." In: *2018 IEEE Conference on Open Systems (ICOS)*. 2018, pp. 25–30. DOI: 10.1109/ICOS.2018.8632815.

[252] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. "Invited Research Overview: End-User Programming." In: *CHI '06 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '06. Montréal, Québec, Canada: ACM, 2006, pp. 75–80. ISBN: 1595932984. DOI: 10.1145/1125451.1125472.

[253] Sandeep Nagar. "IPython." In: *Introduction to Python for Engineers and Scientists: Open Source Solutions for Numerical Computation*. Apress, 2018, pp. 31–45. ISBN: 978-1-4842-3204-0. DOI: 10.1007/978-1-4842-3204-0_3.

[254] Istvan Nagy et al. "VPDSL: A DSL for Software in the Loop Simulations Covering Material Flow." In: *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. ICECCS '12. IEEE Computer Society, 2012, pp. 318–327. ISBN: 9782954181004.

[255] H. Nergaard, N. Ulltveit-Moe, and T. Gjøsæter. "A scratch-based graphical policy editor for XACML." In: *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. 2015, pp. 1–9.

[256] Henrik Nergaard, Nils Ulltveit-Moe, and Terje Gjøsæter. "ViSPE: A Graphical Policy Editor for XACML." In: *Information Systems Security and Privacy*. Springer, 2015, pp. 107–121. ISBN: 978-3-319-27668-7.

[257] Jakob Nielsen and Rolf Molich. "Heuristic Evaluation of User Interfaces." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '90. Seattle, Washington, USA: ACM, 1990, pp. 249–256. ISBN: 0201509326. DOI: 10.1145/97243.97281.

[258] Keith J. O'Hara, Douglas Blank, and James Marshall. "Computational Notebooks for AI Education." In: *Proceedings of the 28th International Florida Artificial Intelligence Research Society Conference* May (2015), pp. 263–268. DOI: 10.13140/2.1.2434.5928.

[259] Keith O'Hara, Douglas Blank, and James Marshall. *Computational Notebooks for AI Education*. 2015. URL: https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS15/paper/view/10349/10312.

[260] Bruno C. d. S. Oliveira and William R. Cook. "Extensibility for the Masses - Practical Extensibility with Object Algebras." In: *ECOOP 2012 – Object-Oriented Programming*. Springer, 2012, pp. 2–27.

[261] Jacqueline Shao Yi Ong et al. "Demo: Expanding Blocks4All with Variables and Functions." In: *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '19. Pittsburgh, PA, USA: ACM, 2019, pp. 645–647. ISBN: 9781450366762. DOI: 10.1145/3308561.3354588.

[262] Jesper Öqvist and Görel Hedin. "Concurrent Circular Reference Attribute Grammars." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 151–162. ISBN: 9781450355254. DOI: 10.1145/3136014.3136032.

[263] Terence Parr. *ANTLR*. https://www.antlr.org/. [Online, accessed 10 October 2021]. 2021.

[264] Terence Parr and Jurgen Vinju. "Towards a Universal Code Formatter through Machine Learning." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 137–151. ISBN: 9781450344470. DOI: 10.1145/2997364.2997383.

[265] E. Pasternak, R. Fenichel, and A. N. Marshall. "Tips for creating a block language with blockly." In: *2017 IEEE Blocks and Beyond Workshop (B B)*. 2017, pp. 21–24.

[266] Evan W. Patton, Michael Tissenbaum, and Farzeen Harunani. "MIT App Inventor: Objectives, Design, and Development." In: *Computational Thinking Education*. Springer, 2019, pp. 31–49. ISBN: 978-981-13-6528-7. DOI: 10.1007/978-981-13-6528-7_3.

[267] Klint Paul. "A Meta-Environment for Generating Programming Environments." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.2 (1993), pp. 176–201. ISSN: 1049-331X. DOI: 10.1145/151257.151260.

[268] Fernando Pérez and Brian E. Granger. "IPython: A System for Interactive Scientific Computing." In: *Computing in Science and Engineering* 9.3 (2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53.

[269] Matthew Pickering, Nicolas Wu, and Csongor Kiss. "Multi-stage programs in context." In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, HaskellICFP 2019, Berlin, Germany, August 18-23, 2019*. ACM, 2019, pp. 71–84. DOI: 10.1145/3331545.3342597.

[270] João Felipe Nicolaci Pimentel et al. "Collecting and Analyzing Provenance on Interactive Notebooks: When IPython Meets No Workflow." In: *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*. TaPP'15. Edinburgh, Scotland: USENIX Association, 2015, p. 10.

[271] João Felipe Pimentel et al. "A Large-scale Study About Quality and Reproducibility of Jupyter Notebooks." In: MSR '19 (2019), pp. 507–517. DOI: 10.1109/MSR.2019.00077.

[272] Gordon D. Plotkin. "A Structural Approach to Operational Semantics." In: *Journal of Logic and Algebraic Programming* 60–61 (2004). Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981, pp. 17–139. DOI: 10.1016/j.jlap.2004.05.001.

[273] Gordon D. Plotkin. "A structural approach to operational semantics." In: *The Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 17–139. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2004.05.001.

[274] Nicolai Pöhner et al. "BlocklySQL: A New Block-Based Editor for SQL." In: *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. WiPSCE'19. Glasgow, Scotland, Uk: ACM, 2019. ISBN: 9781450377041. DOI: 10.1145/3361721.3362104.

[275] Guillaume Pothier, Éric Tanter, and José Piquer. "Scalable omniscient debugging." In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 535–552. DOI: 10.1145/1297105.1297067.

[276] Thomas W. Price and Tiffany Barnes. "Comparing Textual and Block Interfaces in a Novice Programming Environment." In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. Omaha, Nebraska, USA: ACM, 2015, pp. 91–99. ISBN: 9781450336307. DOI: 10.1145/2787622.2787712.

[277] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. "ISnap: Towards Intelligent Tutoring in Novice Programming Environments." In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: ACM, 2017, pp. 483–488. ISBN: 9781450346986. DOI: 10.1145/3017680.3017762.

[278] Thomas W. Price, Rui Zhi, and Tiffany Barnes. "Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior." In: *Artificial Intelligence in Education*. Springer, 2017, pp. 311–322. ISBN: 978-3-319-61425-0.

[279] Roman Rädle et al. "Codestrates: Literate Computing with Webstrates." In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: ACM, 2017, pp. 715–725. ISBN: 9781450349819. DOI: 10.1145/3126594.3126642.

[280] Jonathan Ragan-Kelley et al. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." In: *ACM Transactions on Graphics (TOG)* 31.4 (2012). ISSN: 0730-0301. DOI: 10.1145/2185520.2185528.

[281] Norman Ramsey. "Literate Programming Simplified." In: *IEEE Software* 11.5 (1994), pp. 97–105. ISSN: 0740-7459. DOI: 10.1109/52.311070.

[282] Rascal. *Pico.* 2017. URL: http://tutor.rascal-mpl.org/Recipes/Recipes.html%5C#/Recipes/Languages/Pico/Pico.html.

[283] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. "The Evolution of Forth." In: *The Second ACM SIGPLAN Conference on History of Programming Languages.* HOPL-II. Cambridge, Massachusetts, USA: ACM, 1993, pp. 177–199. ISBN: 0897915704. DOI: 10.1145/154766.155369.

[284] Patrick Rein et al. "Exploratory and Live, Programming and Coding." In: *The Art, Science, and Engineering of Programming* 3.1 (2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2019/3/1.

[285] Achim Reinhardt et al. "Didactic Robotic Fish – An EPSISEP 2016 Project." In: *Interactive Collaborative Learning.* Springer, 2017, pp. 239–253. ISBN: 978-3-319-50337-0.

[286] Thomas W. Reps and Tim" Teitelbaum. "Defining Hybrid Editors with the Synthesizer Generator." In: *The Synthesizer Generator: A System for Constructing Language-Based Editors.* New York, NY: Springer, 1989, pp. 95–142. ISBN: 978-1-4613-9623-9. DOI: 10.1007/978-1-4613-9623-9_6.

[287] Thomas Reps and Tim Teitelbaum. "The Synthesizer Generator." In: SDE 1 (1984), pp. 42–48. DOI: 10.1145/800020.808247.

[288] Mitchel et al. Resnick. "Scratch: Programming for All." In: *Commun. ACM* 52.11 (2009), pp. 60–67. ISSN: 0001-0782.

[289] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages." In: *Proceedings of the ACM Annual Conference - Volume 2.* Boston, Massachusetts, USA, 1972, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852.

[290] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages." In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 363–397. DOI: 10.1023/A:1010027404223.

[291] John C. Reynolds. "Definitional Interpreters Revisited." In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 355–361. DOI: 10.1023/A:1010075320153.

[292] G. Robles et al. "Software clones in scratch projects: on the presence of copy-and-paste in computational thinking learning." In: *2017 IEEE 11th International Workshop on Software Clones (IWSC).* 2017, pp. 1–7. DOI: 10.1109/IWSC.2017.7880506.

[293] F. J. Rodríguez et al. "Toward a Responsive Interface to Support Novices in Block-Based Programming." In: *2019 IEEE Blocks and Beyond Workshop (B B).* 2019, pp. 9–13. DOI: 10.1109/BB48857.2019.8941205.

[294] R V Roque. "OpenBlocks: an extendable framework for graphical block programming systems." In: *Retrieved Dec* (2007). URL: http://hdl.handle.net/1721.1/41550.

[295] Ricarose Vallarta Roque. "OpenBlocks: An Extendable Framework for Graphical Block Programming Systems." MA thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2007.

[296] Simon P. Rose, M.P. Jacob Habgood, and Tim Jay. "Using Pirate Plunder to Develop Children's Abstraction Skills in Scratch." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. Glasgow, Scotland Uk: ACM, 2019, LBW0172:1–LBW0172:6. ISBN: 978-1-4503-5971-9. DOI: 10.1145/3290607.3312871.

[297] Daniel J. Rough and Aaron Quigley. "End-User Development of Experience Sampling Smartphone Apps -Recommendations and Requirements." In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4.2 (2020). DOI: 10.1145/3397307.

[298] Daniel Rough and Aaron Quigley. "Challenges of Traditional Usability Evaluation in End-User Development." In: *End-User Development*. Springer, 2019, pp. 1–17. ISBN: 978-3-030-24781-2.

[299] Arjen Rouvoet et al. "Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages." In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'20)*. 2020, pp. 284–298. ISBN: 9781450370974. DOI: 10.1145/3372885.3373818.

[300] Dashley K. Rouwendal van Schijndel, Audun Stolpe, and Jo E. Hannay. "Using Block-Based Programming and Sunburst Branching to Plan and Generate Crisis Training Simulations." In: *HCI International 2020 - Posters*. Springer, 2020, pp. 463–471. ISBN: 978-3-030-50732-9.

[301] Riemer van Rozen and Tijs van der Storm. "Toward live domain-specific languages." In: *Software & Systems Modeling* 18.1 (2019), pp. 195–212. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0608-7.

[302] Lisa F. Rubin. "Syntax-directed pretty printing–a first step towards a syntax-directed editor." In: *IEEE Transactions on Software Engineering* 2 (1983), pp. 119–127.

[303] Adam Rule. "Design and Use of Computational Notebooks." PhD thesis. University of California San Diego, 2018. URL: https://adamrule.com/files/dissertation/rule%5C_dissertation.pdf.

[304] Adam Rule, Aurélien Tabard, and James D. Hollan. "Exploration and Explanation in Computational Notebooks." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018. ISBN: 9781450356206. DOI: 10.1145/3173574.3173606.

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

[306] Janan Saba, Hagit Hel-Or, and Sharona T. Levy. ""When is the Pressure Zero inside a Container? Mission Impossible": 7th Grade Students Learn Science by Constructing Computational Models Using the Much.Matter.in.Motion Platform." In: *Proceedings of the Interaction Design and Children Conference*. IDC '20. London, United Kingdom: ACM, 2020, pp. 293–298. ISBN: 9781450379816. DOI: 10.1145/3392063.3394442.

[307] Aryan Saini et al. "Aesop: Authoring Engaging Digital Storytelling Experiences." In: *The Adjunct Publication of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: ACM, 2019, pp. 56–59. ISBN: 9781450368179. DOI: 10.1145/3332167.3357114.

[308] Johannes Sametinger. "Literate Programming." In: *Software Engineering with Reusable Components*. Springer, 1997, pp. 211–216. ISBN: 978-3-662-03345-6. DOI: 10.1007/978-3-662-03345-6_18.

[309] Sheeba Samuel and Birgitta König-Ries. "ProvBook: Provenance-based Semantic Enrichment of Interactive Notebooks for Reproducibility." In: *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*. 2018.

[310] Henrique Reinaldo Sarmento et al. "Supporting the Development of Computational Thinking: A Robotic Platform Controlled by Smartphone." In: *Learning and Collaboration Technologies*. Springer, 2015, pp. 124–135. ISBN: 978-3-319-20609-7.

[311] Anthony Savidis and Crystalia Savaki. "Complete Block-Level Visual Debugger for Blockly." In: *Human Systems Engineering and Design II*. Springer, 2020, pp. 286–292. ISBN: 978-3-030-27928-8.

[312] B. Selwyn-Smith et al. "Co-located Collaborative Block-Based Programming." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 107–116. DOI: 10.1109/VLHCC.2019.8818895.

[313] Mazyar Seraj et al. "Smart Homes Programming: Development and Evaluation of an Educational Programming Application for Young Learners." In: *Proceedings of the 18th ACM International Conference on Interaction Design and Children*. IDC '19. Boise, ID, USA: ACM, 2019, pp. 146–152. ISBN: 978-1-4503-6690-8. DOI: 10.1145/3311927.3323157.

[314] Amazon Web Services. *AWS CloudFormation Documentation*. https://go.aws/3ENgF1k. [Online, accessed 12 July 2021]. 2021.

[315] Teddy Seyed et al. "MakerArcade: Using Gaming and Physical Computing for Playful Making, Learning, and Creativity." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. Glasgow, Scotland Uk: ACM, 2019, LBW0174:1–LBW0174:6. ISBN: 978-1-4503-5971-9. DOI: 10.1145/3290607.3312809.

[316] J. C. Shaw. "JOSS: A designer's view of an experimental on-line computing system." In: *AFZPS Conference Proceedings, vol. 26, 1964 Fall Joint Computer Conference*. 1964, pp. 455–464.

[317] Helen Shen. "Interactive Notebooks." In: *Nature* 515 (2014), pp. 151–152.

[318] D. Shepherd et al. "[Engineering Paper] An IDE for Easy Programming of Simple Robotics Tasks." In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2018, pp. 209–214. DOI: 10.1109/SCAM.2018.00032.

[319] Yasin N. Silva and Jaime Chon. "DBSnap: Learning Database Queries by Snapping Blocks." In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 179–184. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677220.

[320] Yasin N. Silva et al. "DBSnap++: Creating Data-driven Programs by Snapping Blocks." In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018. Larnaca, Cyprus: ACM, 2018, pp. 170–175. ISBN: 978-1-4503-5707-4. DOI: 10.1145/3197091.3197114.

[321] Charles Simonyi, Magnus Christerson, and Shane Clifford. "Intentional Software." In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 451–464. ISBN: 1595933484. DOI: 10.1145/1167473.1167511.

[322] Cynthia Solomon et al. "History of Logo." In: *Proc. ACM Program. Lang.* 4.HOPL (2020). DOI: 10.1145/3386329.

[323] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. "Testing Scratch Programs Automatically." In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 165–175. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338910.

[324] William Stein and David Joyner. "SAGE: system for algebra and geometry experimentation." In: *ACM SIGSAM Bulletin* (2005). ISSN: 0163-5824. DOI: 10.1145/1101884.1101889.

[325] Tijs van der Storm. *Rascal QL - Tutorial*. 2019. URL: https://bit.ly/3HlHwTM (visited on 12/22/2019).

[326] Tijs van der Storm. *Salix*. 2019. URL: https://github.com/cwi-swat/salix (visited on 12/22/2019).

[327] Andrew Stratton, Chris Bates, and Andy Dearden. "Quando: Enabling Museum and Art Gallery Practitioners to Develop Interactive Digital Exhibits." In: *End-User Development*. Springer, 2017, pp. 100–107. ISBN: 978-3-319-58735-6.

[328] Glenn Strong, Sean O'Carroll, and Nina Bresnihan. "A Block Based Editor for Python." In: *Proceedings of the 13th Workshop in Primary and Secondary Computing Education*. WiPSCE '18. Potsdam, Germany: ACM, 2018, 30:1–30:2. ISBN: 978-1-4503-6588-8. DOI: 10.1145/3265757.3265788.

[329] C. J. Sutherland and B. A. MacDonald. "NaoBlocks: A Case Study of Developing a Children's Robot Programming Environment." In: *2018 15th International Conference on Ubiquitous Robots (UR)*. 2018, pp. 431–436. DOI: 10.1109/URAI.2018.8441843.

[330] Ryo Suzuki et al. "Implementing Node-Link Interface into a Block-Based Visual Programming Language." In: *Human-Computer Interaction. Interaction in Context*. Springer, 2018, pp. 455–465. ISBN: 978-3-319-91244-8.

[331] A. Swidan, A. Serebrenik, and F. Hermans. "How do Scratch Programmers Name Variables and Procedures?" In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2017, pp. 51–60. DOI: 10.1109/SCAM.2017.12.

[332] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. "Designing and implementing combinator languages." In: *Advanced Functional Programming*. Springer, 1999, pp. 150–206. DOI: 10.1007/10704973_4.

[333] A. G. Tamilias et al. "B@SE: Blocks for @rduino in the Students' educational process." In: *2017 IEEE Global Engineering Education Conference (EDUCON)*. 2017, pp. 910–915. DOI: 10.1109/EDUCON.2017.7942956.

[334] Steven L. Tanimoto. "A perspective on the evolution of live programming." In: *1st International Workshop on Live Programming (LIVE'13)*. IEEE. 2013, pp. 31–34. DOI: 10.1109/LIVE.2013.6617346.

[335] Karen Tatarian et al. "Tailoring a ROS Educational Programming Language Architecture." In: *Robotics in Education*. Springer, 2019, pp. 217–229. ISBN: 978-3-319-97085-1.

[336] S. Taylor et al. "Position: IntelliBlox: A Toolkit for Integrating Block-Based Programming into Game-Based Learning Environments." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 55–58. DOI: 10.1109/BB48857.2019.8941222.

[337] The Haml Team. *HAML*. URL: http://haml.info/ (visited on 12/22/2019).

[338] P. Techapalokul and E. Tilevich. "Understanding recurring quality problems and their impact on code sharing in block-based software." In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pp. 43–51. DOI: 10.1109/VLHCC.2017.8103449.

[339] P. Techapalokul and E. Tilevich. "Code Quality Improvement for All: Automated Refactoring for Scratch." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2019, pp. 117–125. DOI: 10.1109/VLHCC.2019.8818950.

[340] P. Techapalokul and E. Tilevich. "Position: Manual Refactoring (by Novice Programmers) Considered Harmful." In: *2019 IEEE Blocks and Beyond Workshop (B B)*. 2019, pp. 79–80. DOI: 10.1109/BB48857.2019.8941201.

[341] Warren Teitelman. "PILOT: A Step Toward Man-Computer Symbiosis." PhD thesis. MIT, 1966. URL: http://hdl.handle.net/1721.1/6905.

[342] Warren Teitelman. "History of Interlisp." In: *Celebrating the 50th Anniversary of Lisp*. Nashville, Tennessee: ACM, 2008. ISBN: 9781605583839. DOI: 10.1145/1529966.1529971.

[343] Marilyn Tenorio Melenje María et al. "Debugging Block-Based Programs." In: *Human-Computer Interaction*. Springer, 2019, pp. 98–112. ISBN: 978-3-030-05270-6.

[344] Ulyana Tikhonova et al. "Constraint-based Run-time State Migration for Live Modeling." In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2018. Boston, MA, USA: ACM, 2018, pp. 108–120. ISBN: 978-1-4503-6029-6. DOI: 10.1145/3276604.3276611.

[345] J. Trenouth. "A Survey of Exploratory Software Development." In: *The Computer Journal* 34.2 (1991), pp. 153–163. ISSN: 0010-4620. DOI: 10.1093/comjnl/34.2.153.

[346] Franklyn Turbak et al. "Events-first Programming in APP Inventor." In: *J. Comput. Sci. Coll.* 29.6 (2014), pp. 81–89. ISSN: 1937-4771.

[347] Tommaso Turchi, Daniela Fogli, and Alessio Malizia. "Fostering computational thinking through collaborative game-based learning." In: *Multimedia Tools and Applications* 78.10 (2019), pp. 13649–13673. ISSN: 1573-7721. DOI: 10.1007/s11042-019-7229-9.

[348] Tommaso Turchi, Alessio Malizia, and Alan Dix. "TAPAS: A tangible End-User Development tool supporting the repurposing of Pervasive Displays." In: *Journal of Visual Languages & Computing* 39 (2017), pp. 66–77. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2016.11.002.

[349] Phil Turner and Susan Turner. "Supporting Cooperative Working Using Shared Note-books." In: *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work*. Springer, 1997, pp. 281–295. ISBN: 978-94-015-7372-6. DOI: 10.1007/978-94-015-7372-6_19.

[350] Adilson Vahldick, António José Mendes, and Maria José Marcelino. "Learning Analytics Model in a Casual Serious Game for Computer Programming Learning." In: *Serious Games, Interaction and Simulation*. Springer, 2017, pp. 36–44. ISBN: 978-3-319-51055-2.

[351] Tijs van der Storm. "Semantic deltas for live DSL environments." In: *1st International Workshop on Live Programming (LIVE)*. IEEE, 2013, pp. 35–38. ISBN: 978-1-4673-6265-8. DOI: 10.1109/LIVE.2013.6617347.

[352] A. van Deursen, P. Klint, and F. Tip. "Origin tracking." In: *Journal of Symbolic Computation* 15.5 (1993), pp. 523–545. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(06)80004-0.

[353] Eric Van Wyk et al. "Forwarding in Attribute Grammars for Modular Language Design." In: *Compiler Construction*. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 128–142. ISBN: 978-3-540-43369-9. DOI: 10.1007/3-540-45937-5_11.

[354] Michael L. Van de Vanter, Marat Boshernitsan, and San Antonio Avenue. "Displaying and Editing Source Code in Software Engineering Environments." In: 2000.

[355] Mauricio Verano Merino and Tijs van der Storm. "Block-Based Syntax from Context-Free Grammars." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: ACM, 2020, pp. 283–295. ISBN: 9781450381765. DOI: 10.1145/3426425.3426948.

[356] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: A Language Parametric Notebook Generator (Tool Demo)." In: SLE 2018 (2018), pp. 210–214. DOI: 10.1145/3276604.3276981.

[357] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. *Bacata*. Version 1.0.0. 2020. DOI: 10.5281/zenodo.3636179.

[358] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. *Bacata-demos*. Version 1.0.0. 2020. DOI: 10.5281/zenodo.3636103.

[359] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. "Bacatá: Notebooks for DSLs, Almost for Free." In: *The Art, Science, and Engineering of Programming* 4.3 (2020). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/11.

[360] Mauricio Verano Merino et al. "Getting Grammars into Shape for Block-Based Editors." In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2021. Virtual, USA: ACM, 2021. ISBN: 978-1-4503-9111-5/21/10. DOI: 10.1145/3486608.3486908.

[361] Jacques Verriet et al. "Model-Driven Development of Logistic Systems Using Domain-Specific Tooling." In: *Complex Systems Design & Management*. Springer, 2013, pp. 165–176. ISBN: 978-3-642-34404-6.

[362] R. Vinayakumar, K. Soman, and P. Menon. "Alg-Design: Facilitates to Learn Algorithmic Thinking for Beginners." In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–6. DOI: 10.1109/ICCCNT.2018.8493952.

[363]   R. Vinayakumar, K. Soman, and P. Menon. "CT-Blocks: Learning Computational Think-ing by Snapping Blocks." In: *2018 9th International Conference on Computing, Communi-cation and Networking Technologies (ICCCNT)*. 2018, pp. 1–7. DOI: 10.1109/ICCCNT.2018.8493669.

[364]   R. Vinayakumar, K. Soman, and P. Menon. "DB-Learn: Studying Relational Algebra Concepts by Snapping Blocks." In: *2018 9th International Conference on Computing, Com-munication and Networking Technologies (ICCCNT)*. 2018, pp. 1–6. DOI: 10.1109/ICCCNT.2018.8494181.

[365]   R. Vinayakumar, K. Soman, and P. Menon. "Fractal Geometry: Enhancing Computa-tional Thinking with MIT Scratch." In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–6. DOI: 10.1109/ICCCNT.2018.8494172.

[366]   R. Vinayakumar, K. Soman, and P. Menon. "Map-Blocks: Playing with Online Data and Infuse to Think in a Computational Way." In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–6. DOI: 10.1109/ICCCNT.2018.8493700.

[367]   Markus Voelter et al. "Efficient Development of Consistent Projectional Editors Using Grammar Cells." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 28–40. ISBN: 9781450344470. DOI: 10.1145/2997364.2997365.

[368]   P. Voštinár. "Programming LEGO EV3 in Microsoft MakeCode." In: *2020 IEEE Global En-gineering Education Conference (EDUCON)*. 2020, pp. 1868–1872. DOI: 10.1109/EDUCON45650.2020.9125170.

[369]   Premysl Vysoký, Pavel Parízek, and Václav Pech. *INGRID: Creating Languages in MPS from ANTLR Grammars*. Tech. rep. D3S-TR-2018-01. Department of Distributed and Dependable Systems, Charles University, 2018, pp. 1–18.

[370]   Theodore W. Gray and Stephen Wolfram. "Method and system for presenting input expressions and evaluations of the input expressions on a workspace of a computational system." 2013.

[371]   Aditi Wagh and Uri Wilensky. "EvoBuild: A Quickstart Toolkit for Programming Agent-Based Models of Evolutionary Processes." In: *Journal of Science Education and Technology* 27.2 (2018), pp. 131–146. ISSN: 1573-1839. DOI: 10.1007/s10956-017-9713-1.

[372]   Michał Walicki and Sigurd Meldal. "Algebraic Approaches to Nondeterminism – an Overview." In: *ACM Computing Surveys* 29.1 (1997), pp. 30–81. ISSN: 0360-0300. DOI: 10.1145/248621.248623.

[373]   Wengran Wang et al. "Crescendo: Engaging Students to Self-Paced Programming Practices." In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: ACM, 2020, pp. 859–865. ISBN: 9781450367936. DOI: 10.1145/3328778.3366919.

[374]   Martin Ward. "Language Oriented Programming." In: *Software-Concepts & Tools* 15 (1994), pp. 147–161. DOI: 10.1007/978-1-4302-2390-0_12.

[375]   D. Weintrop et al. "Blockly goes to work: Block-based programming for industrial robots." In: *2017 IEEE Blocks and Beyond Workshop (B B)*. 2017, pp. 29–36. DOI: `10.1109/BLOCKS.2017.8120406`.

[376]   David Weintrop. "Block-based Programming in Computer Science Education." In: *Commun. ACM* 62.8 (2019), pp. 22–25. ISSN: 0001-0782. DOI: `10.1145/3341221`.

[377]   David Weintrop and Uri Wilensky. "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*. IDC '15. Boston, Massachusetts: ACM, 2015, pp. 199–208. ISBN: 9781450335904. DOI: `10.1145/2771839.2771860`.

[378]   David Weintrop and Uri Wilensky. "Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Stanford, California, USA: ACM, 2017, pp. 183–192. ISBN: 9781450349215. DOI: `10.1145/3078072.3079715`.

[379]   David Weintrop and Uri Wilensky. "How block-based, text-based, and hybrid block/text modalities shape novice programming practices." In: *International Journal of Child-Computer Interaction* 17 (2018), pp. 83–92. ISSN: 2212-8689. DOI: `10.1016/j.ijcci.2018.04.005`.

[380]   David Weintrop and Uri Wilensky. "How block-based, text-based, and hybrid block/text modalities shape novice programming practices." In: *International Journal of Child-Computer Interaction* 17 (2018), pp. 83–92. ISSN: 22128689. DOI: `10.1016/j.ijcci.2018.04.005`.

[381]   David Weintrop and Uri Wilensky. "Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms." In: *Computers & Education* 142 (2019), p. 103646. ISSN: 0360-1315. DOI: `10.1016/j.compedu.2019.103646`.

[382]   David Weintrop et al. "Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 366:1–366:12. ISBN: 978-1-4503-5620-6. DOI: `10.1145/3173574.3173940`.

[383]   David Weintrop et al. "Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18* (2018), pp. 1–12. DOI: `10.1145/3173574.3173940`.

[384]   David S. Wile. "Abstract Syntax from Concrete Syntax." In: *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997*. Ed. by W. Richards Adrion et al. ACM, 1997, pp. 472–480. DOI: `10.1145/253228.253388`.

[385]   MICHELLE WILKERSON-JERDE, ADITI WAGH, and URI WILENSKY. "Balancing Curricular and Pedagogical Needs in Computational Construction Kits: Lessons From the DeltaTick Project." In: *Science Education* 99.3 (2015), pp. 465–499. DOI: `10.1002/sce.21157`.

[386]   Manuel Wimmer and Gerhard Kramler. "Bridging grammarware and modelware." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2005, pp. 159–168.

[387] Stephen Wolfram. *The Mathematica Book (4th Edition)*. Cambridge University Press, 1999. ISBN: 0521643147.

[388] Yihui Xie. "knitr: A General-Purpose Tool for Dynamic Report Generation in R." In: 8.1 (2013), pp. 1–12. URL: https://bit.ly/3HsB5ye.

[389] Yihui Xie, Joseph J. Allaire, and Garrett Grolemund. *R Markdown: The Definitive Guide*. 2018. ISBN: 1439144834. DOI: 10.1016/B978-0-12-814447-3.00041-0.

[390] Stelios Xinogalos, Maya Satratzemi, and Christos Malliarakis. "Microworlds, games, animations, mobile apps, puzzle editors and more: What is important for an introductory programming environment?" In: *Education and Information Technologies* 22.1 (2017), pp. 145–176. ISSN: 1573-7608. DOI: 10.1007/s10639-015-9433-1.

[391] Susan A. Yoon, Jooeun Shim, and Noora Noushad. "Trade-Offs in Using Mobile Tools to Promote Scientific Action with Socioscientific Issues." In: *TechTrends* 63.5 (2019), pp. 602–610. ISSN: 1559-7075. DOI: 10.1007/s11528-019-00408-z.

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

[393] Yuchen Zhao et al. "Structured and uncertainty-aware data storytelling." In: *CHI-19 Workshop on HCI for Accurate, Impartial and Transparent Journalism: Challenges and Solutions*. 2019. URL: https://eprints.soton.ac.uk/431822/.

[394] Sharon Zhou et al. "Ingenium: Engaging Novice Students with Latin Grammar." In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: ACM, 2016, pp. 944–956. ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858239.

[395] Abigail Zimmermann-Niefield et al. "Youth Making Machine Learning Models for Gesture-Controlled Interactive Media." In: *Proceedings of the Interaction Design and Children Conference*. IDC '20. London, United Kingdom: ACM, 2020, pp. 63–74. ISBN: 9781450379816. DOI: 10.1145/3392063.3394438.

[396] Hubert Zimmermann. "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection." In: *IEEE Transactions on Communications* 28.4 (1980), pp. 425–432. ISSN: 0090-6778. DOI: 10.1109/TCOM.1980.1094702.

CURRICULUM VITAE

Mauricio Verano Merino was born on the 7th of December of 1990 in Bogotá, Colombia. Mauricio decided to study Systems and Computer Engineering at Universidad de Los Andes (Bogotá, Colombia), because he wanted to explore how to create new ways of interacting with computers. When he graduated in 2013, the university had the possibility to start a partnership with Heinsohn Business Technology through a project on how to migrate on-premise enterprise web applications to the cloud. Prof. dr. Rubby Casallas and dr. Kelly Garces brought Mauricio on board to settle the partnership where he worked as a Graduate Research Assistant. Here he learned how to understand, (re)design and (re)build software to adapt it into new environments. When he finished in 2014, Mauricio had a chance to be a lecturer at the university and so his career in academia started. After a while, in 2016, Eindhoven University of Technology had a PhD position to work initially in collaboration with Canon Production Printing (former Océ) and later on with Siemens Digital Industries Software. Mauricio started working there under the supervision of prof.dr. Jurgen Vinju, prof. dr. Mark van den Brand, prof.dr. Tijs van der Storm, dr. Lou Sommers, and dr.Ing Mike Nicolai. During these five years he explored how to offer different user interfaces for Domain-Specific Languages to increase their adoption in different contexts. This was when everything came to place and he managed to understand, (re)design and (re)build software structures to create an accessible way of interacting with computers through programming. All these explorations, paths and discoveries, everything he found during this time is contained here and is presented in this dissertation.

After finishing his PhD, Mauricio will continue his journey in academia as an Assistant professor at the Vrije Universiteit Amsterdam.

IPA DISSERTATION SERIES

Titles in the IPA Dissertation Series since 2019

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D.Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VUA. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04