Suppose You Had Blocks within a Notebook

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

Juan Pablo Sáenz
juan.saenz@polito.it
Politecnico di Torino
Turin, Italy

Ana María Díaz Castillo
anamdiaz9328@gmail.com
Teach for All
Bogotá, Colombia

Abstract

Computational notebooks have been gaining prominence as a development environment suitable for non-experienced developers. However, it requires proficiency in writing syntactically and semantically correct code. In this article, we propose integrating a block-based approach into computational notebooks to prevent syntactical errors and ease the non-expert developers' adoption. Furthermore, we rely on two tools previously implemented (Bacatá and Kogi) to (i) create a computational notebook for Domain-Specific Languages and (ii) generate a block-based representation upon the language definition. Consequently, our approach does not exclusively focus on integrating a block-based environment into computational notebooks but on enabling the creation and integration of domain-specific block-based environments into notebooks. Future work concerns the evaluation of our proposal through a user study.

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

Keywords: Block-based languages, Scratch, Computational notebook, Documentation, Jupyter, DSLs, Blockly

ACM Reference Format:
Mauricio Verano Merino, Juan Pablo Sáenz, and Ana María Díaz Castillo. 2022. Suppose You Had Blocks within a Notebook. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3563836.3568728

1 Introduction

End-User Programming (EUP) is a field of the Human-Computer Interaction (HCI) domain that studies the methods and technologies that enable people to customize their software [18]. Such customization can be achieved at different levels and involve diverse artifacts such as programming languages, Integrated Development Environments (IDEs), Read-Eval-Print Loop (REPLs), development frameworks, Tangible User Interfaces (TUIs), and Graphical User Interfaces (GUIs).

Domain-Specific Languages (DSLs) are small languages tailored to specific problems in a concrete domain. They use domain concepts rather than programming concepts [21, 32]. In this manner, DSLs provide domain experts with a higher level of abstraction that enable them to write their software. Due to this, DSLs have been gaining prominence [5], increasing the number of non-programmers that are developing software [27]. DSLs and their tooling are created by language engineers through *Language Workbenches* (LWB) [8, 9], while computational notebooks have allowed non-experienced developers to write, document, and execute their code within a single document. For this reason, they have gained popularity among data scientists, journalists, and statisticians.

In this scenario, this paper presents Noteblocks, a computational notebook that uses block-based syntax as a graphical input to write code. The advantages of Block-based-environments are well known (e.g., preventing syntax errors and reducing implementation times). Taking that into account, our proposal's novelty lies in generating the block-based representation of a DSL and integrating it into a computational notebook environment. This will be done almost for free and will rely entirely in the language definition.

2 Context

This section briefly describes computational notebooks and Block-based environments. It presents two tools regarding DSLs, one that integrates them with computational notebooks, and other one that generates a block-based interface for them.

2.1 Computational Notebooks

Computational notebooks are cell-based documents that allow users to interleave documentation and code. Figure 1 shows an example of a notebook in which the first cell (*documentation cell*) contains prose written in Markdown; the second cell (*code cell*) contains executable Python code, and below the code cell it shows the corresponding result (*output cell*) of running the code. In general, notebooks have lowered the entry barrier to programming for novices [28]. Nowadays there are more than 60 notebook platforms [17] and one of the most popular is the Jupyter project [16] with

PAINT '22, December 05, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9910-4/22/12.
https://doi.org/10.1145/3563836.3568728

millions of users [29]. Computational notebooks have become a prominent End-user programming tool where users can quickly run and test their prototypes using code in an exploratory programming setting. However, most notebook platforms have built-in support for popular general-purpose languages (e.g., Python, JavaScript, and R, among others), and adding support for new languages in them, even if possible, is time-consuming and cumbersome.

Prints the line "Hello PAINT 2022"

```
[1]: print("Hello PAINT 2022")
     Hello PAINT 2022
```

Figure 1. Example of a cell execution in Jupyter computational notebook

2.2 Block-Based Environments

Block-based environments are visual programming environments that use jigsaw-like elements to represent language constructs; in them, code constructs are manipulated as pieces of a puzzle. One of the most popular and successful Block-based environments is Scratch [26], an educational tool for teaching children how to program. Indeed, thanks to their friendly, intuitive graphical interface and interaction design, they have become a prominent alternative to lower the entry barrier to programming [2]. They offer a what-you-see-is-what-you-get (WYSIWYG) experience, which means that the developer or the end-user does not need to know the complete syntax of the language in order to program. Although these environments require some knowledge of the code constructs to build a program, such knowledge is less specialized and more suitable for non-technical users [1, 6]. Nevertheless, for the computational notebooks, supported languages are limited, and creating new block-based environments requires significant effort and technical proficiency. For DSLs, it is even more challenging due to the small development teams. Figure 2 shows a block-based language developed with Google Blockly, a client-side library for creating block-based environments.

2.3 Kogi and Bacatá

Kogi and Bacatá are tools that allow language engineers to reuse existing language definitions to generate programming environments for DSLs in a generic fashion. On the one hand, Kogi [34] is a tool for describing and deriving block-based environments from context-free grammars using the Rascal [15] Language Workbench (LWB) and Blockly. Kogi performs five operations: takes a DSL written in Rascal's built-in grammar formalism; preprocesses it to eliminate disambiguation constructs and inline chain rules; maps the

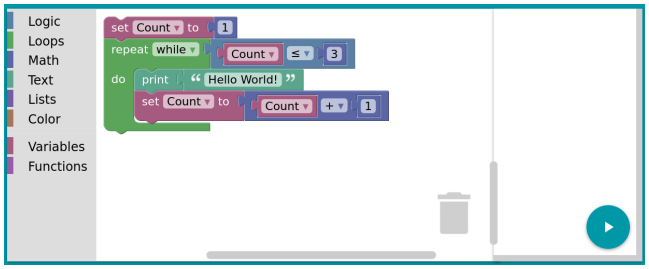

Figure 2. Block-based editor created using Google Blockly

grammar to the constructs available in the Blocky environment; customizes the appearance of the Blocks based on the preferences defined by the language engineer; and generates the corresponding Blocky representation. One of Kogi's key strengths is that it reuses existing language definitions to offer a new UI for the language almost for free.

On the other hand, Bacatá [20] is a language-parametric tool for generating notebook interfaces for DSLs. It takes as input the syntax definition of the language expressed as context-free grammar and a Read-Eval-Print Loop (REPL) interpreter definition to generate a Jupyter kernel. This kernel is generated by reusing existing language components such as syntaxes (concrete and abstract), parsers, type checkers, code generators, and interpreters. In this manner, Bacatá allows language engineers to offer their users the possibility to use their DSL within a Jupyter notebook, using code cells or command line scripts managed by the REPL.

These two approaches are possible by using meta- programming tools and techniques [23], and are part of an existing research topic on programming environment generation [3, 7, 13, 25, 31]. When used together, Kogi and Bacatá give rise to Noteblocks, which is our approach to generate a block-based interface for a DSL that can be executed within a computational notebook environment. In the next section, we will describe the Noteblocks architecture and implementation in detail.

3 Noteblocks

As mentioned in the previous section, Computational notebooks have become a prominent End-user programming tool where users can quickly run and test their code prototypes in an exploratory programming setting. However, using them requires semantically and syntactically proficiency in the chosen programming language. Additionally, just a small set of general purpose programming languages are supported, and integrating a language is technically challenging and time spending.

Visual languages, as block-based environments, provide a friendly and intuitive graphical interface through which users can implement code without facing syntax issues. One of their major benefits is that developers do not need to

memorize the language's syntax since they are always available in the block-based editor toolbox. Consequently, this characteristic has significantly lowered the entry barriers to programming.

In this scenario, our proposal aims to merge the best of both worlds which is to create a cost-efficient tool to support using DSLs in a widely adopted exploratory programming setup. For this purpose, Noteblocks proposes a generative approach that builds upon previous work where the language definition is used as input to generate and reuse existing language components. In this section, we describe our solution architecture and how the integration with the Jupyter front-end was implemented. Then we illustrate a running example.

3.1 Overall Architecture

Noteblocks has three main components: Kogi, Bacatá, and a Jupyter front-end extension. While Kogi and Bacatá are implemented using Rascal, the front-end extension is implemented within the Jupyter development environment using JavaScript. Figure 3 shows Noteblocks's general architecture.

As outlined in the previous section, the initial input artifacts are the language definition (grammar) and a REPL interpreter. Kogi takes the language definition to derive a Block-Based Editor (marked as *BBE* in Figure 3). Bacatá, apart from the definition, uses the definition of an interpreter, expressed as a REPL interface, to generate the Jupyter Kernel. In this regard, language engineers can reuse an existing interpreter or develop a new one[1]. Also, sequential languages [2] are preferred to generate the kernel.

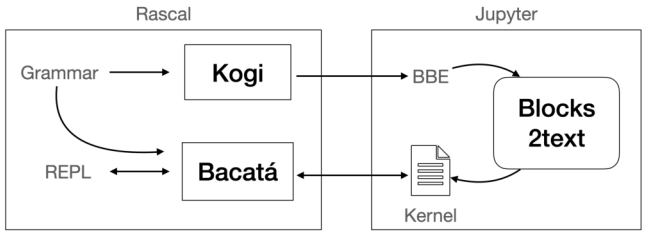

Figure 3. Noteblocks's architecture

The *Blocks2text* module transforms the block-based programs into a textual representation suitable to be parsed by the corresponding language parser. Without this transformation, communication between Kogi and Bacatá would not be possible.

3.2 Jupyter's Notebook Front-End Extension

The implementation of Noteblocks was achieved upon customizing the Jupyter notebook front-end in two ways: first, by adding block-based support, and second, by executing block-based programs. To that end, a Jupyter extension was developed in which a button called Tile view (as shown in Figure 4) was graphically added on top of code cells. By clicking that button, the representation of the code in the cell switches from text-based to block-based and vice-versa. From the development point of view, the block-based editing support was implemented through Google Blockly [10], embedding it into the Jupyter notebook code cell view. These changes allowed Noteblocks to read Kogi's generated block-based environment and render it as part of the input cell within a notebook.

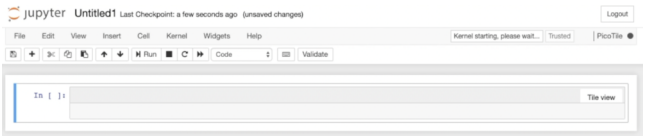

Figure 4. Block-based editor being enabled within a Jupyter notebook using Noteblocks

As described in the previous subsection, the *Blocks2text* module is fundamental in transforming the block-based programs into text, so that the language's interpreter can be completely reused. In this way, the text is sent back to the language's interpreter in Rascal, that knows how to execute the code snippet and what to retrieve back to the graphical interface. This feature is interesting because language engineers do not need to have different versions of the interpreter depending on the front-end (text-based or block-based); a single interpreter works for both UIs.

3.3 Running Example: Pico

To evaluate our approach, we used Pico: a small programming language (similar to the While language [22]) commonly used to teach programming languages semantics. Therefore, we reused an existing Rascal implementation of Pico [24]. Based on this implementation, Noteblocks received the grammar as input, and Kogi generated the block-based environment. Then, using the grammar and the REPL, Bacatá generated the Jupyter kernel. In this process, Noteblocks was responsible for loading the block-based environment into Jupyter when the notebook started. Technically speaking, the block-based environment is a JavaScript file containing the definition of the blocks and their associated toolbox.

Once the components were deployed and executed, a Pico notebook was opened. In the Jupyter default front-end interface, a code cell was added to every code cell, as described in the previous subsection. Figure 5 shows the Pico program implemented using the block-based notation within a code cell in the Jupyter computational notebook. Additionally, a full video on how the environment looked like is available online [3]. After the users executed the code cell, the output

[1] Development of REPL-interfaces is part of the language design, and there are some crucial considerations when deriving a REPL interpreter [30]

[2] A sequential language is a language in which the concatenation of two valid programs is also a valid program [30]

[3] https://surfdrive.surf.nl/files/index.php/s/UVzhV2QcEfj4Lf6

was displayed in the default output cell as when executing text-based code snippets.

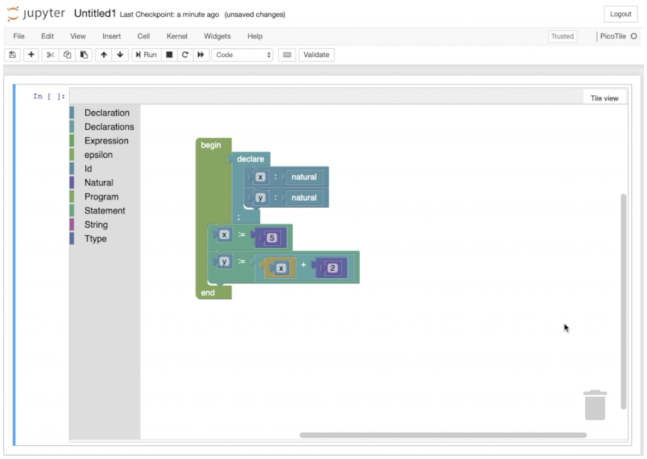

Figure 5. Block-based program implemented within a Jupyter notebook using Noteblocks

4 Discussion and Future Work

End-user programminghas received much attention in the last decade for empowering non-technical people (e.g., end-users and novice users) from different domains and backgrounds to develop software. Since the number of end-users significantly outnumber professional programmers [27], the impact of EUP could be huge.

Block-based environments provide a good interface for end-users to experiment with a language and to focus on what their programs should do, instead of their syntax. However, Block-based interfaces might not be effective for experienced users. Also, this metaphor might not be suited for all languages as discussed by Verano et al.[33], but more research is needed to determine what languages are the most appropriate to offer this type of interface. One important element is that thanks to the use of language workbenches, Noteblocks offers an alternative programming environment to the traditional textual IDE settings. This means, that language engineers have a single language definition and they obtain almost for free notebooks with block-based notation. If the block-based notation is not suitable for the language, engineers can use only Bacatáto obtain a notebook environment.

One of the limitations of the current approach is that each code cell contains a block-based workspace, which might have consequences in terms of browser's performance when the number of code cells increases. Moreover, from the usability point of view, the current approach is not optimal since the visual nature of the block-based editors requires more space than text-based code editors. Therefore, further research is required to evaluate the current approach and find solutions to address this issue without redesigning the whole notebook experience.

5 Related Work

Computational notebooks involve diverse domains such as end-user programming, exploratory programming, live programming, and literate computing. This section presents research on the customization and adoption of computational notebooks in diverse domains and contexts.

Kery *et al.* [14] extended computational notebooks with a new API that enables new forms of interactive graphical data tools that seamlessly translate to code. Therefore, these tools can represent themselves as both code and GUI as needed. Specifically, the authors implemented six example notebook widgets (`table`, `plot`, `image`, `confusion`, `api`, `datasplit`, and `save`) to support data scientists on tasks ranging from machine learning to visualization support. This approach aims to provide richer interactions and more context-aware support tailored to a data analyst's specific workflow. Furthermore, a study conducted with nine professionals evidenced that the GUI makes suggested operations understandable to novice users, who may not yet be familiar with machine learning best practices. This work is particularly relevant to our proposal because it integrates the hybrid editor concept into the computational notebook environment. Indeed, Noteblocksis partially motivated by the author's willingness to see more HCI systems work in these intersections to support data workers and programmers in the coming years.

In the context of the Jupyter framework, Verano *et al.* [35] developed a language parametric notebook generator for Domain-Specific Languages (DSL). The authors aimed to enable language engineers to quickly implement Jupyter language kernels for their DSLs by reusing, as much as possible, existing language components, such as parsers, code generators, and Read-Eval-Print Loops (REPLs). By providing generic hooks for registering language services, the authors could hide the low-level complexity of Jupyter's wire protocol. Thus, implementing a notebook interface for a DSL becomes a matter of writing a few lines of code. In the same vein, Verano *et al.* [19] has recently discussed the possibility of defining widgets within computational notebooks to provide direct insights into the program state graphically. The authors present an execution graph widget and a variable watcher widget. The first widget enables users to navigate the results (configurations) obtained after executing the notebook cells, and the second enables them to reveal the entire execution state and history to the user.

Head *et al.* [11] introduce a set of code gathering tools and extensions to computational notebooks aimed for helping analysts to find, clean, recover, and compare versions of code in cluttered inconsistent notebooks. After conducting a qualitative usability study with 12 experienced analysts, the authors

determined that their presented tools were found helpful in cleaning notebooks and generating personal documentation and lightweight versioning.

Yin *et al.* [36] describes CyberGIS-Jupyter, a framework for performing data-intensive, reproducible, and scalable geospatial analytics using the Jupyter Notebook. The authors' goal was to reach agility and reproducibility in geospatial analytics. In their proposal, instead of developing customized and web-based GUI interfaces that require professional skills, the authors relied on using a Jupyter notebook as a GUI development platform. Specifically, they developed a set of utilities to support CyberGIS operations, using a Jupyter Interactive Widgets library. Furthermore, the authors used container virtualization technologies to record and reproduce computational environments with the exact versions of all external libraries. This way, the framework allows researchers to share and build on each other's work in a large-scale geospatial analytics setup.

Chen *et al.* [4] presents an approach of mixing graphical user interfaces for defining image processing pipelines with computational notebooks. This approach uses node diagrams as visual input for the code cells of the notebook. However, this approach does not offer a generic solution for other languages to use this notation. Similarly, Homer and Noble [12] introduced Tiled Grace, a hybrid editor that allows users to edit their programs using blocks or text in the same environment; however, this approach is not integrated within a computational notebook setting nor offers a generic mechanism to be reused by other languages.

6 Conclusion

Noteblocks shows that it is possible to merge both approaches, namely computational notebooks, and block-based environments. The result is a single language parametric tool that aims to improve the end-users experience; all this with a low effort for developers, thanks to the generative approach. Also, as shown in this article, Noteblocks allows language engineers to quickly prototype and test new tools. This allows DSLs to keep the pace of new tooling commonly developed only for popular General Purpose Programming Languages (GPLs). Thanks to the generative approach, developers can get a notebook with blocks as input almost for free. The effort required to use Noteblocks is minimal if the language's definition and its interpreter are already defined, and the latter belongs to the class of sequential languages. Finally, offering more friendly tooling for end-users not only improves the productivity of end-users but also improves the adoption of DSLs by offering popular tools commonly available only for popular GPLs that have proven to reduce the entry barrier to programming for DSLs.

Computational notebooks are popular among different communities, such as professional developers, data scientists, domain experts, and novice programmers, because they have lowered the entry barrier to programming compared to traditional programming environments (professional IDEs) requiring a compile-edit-run loop. However, it is still possible to enrich the end-user's programming experience by enabling different notations for creating programs (e.g., using GUIs together with code). Moreover, it is crucial to help users create programs, but it is also essential to help users understand their program execution cycles. Therefore, in future work, we want also to study how to offer better feedback to end-users so that they can achieve their tasks and reduce their cognitive load by revealing the underlying system's state.

References

[1] Austin Cory Bart, Javier Tibau, Dennis Kafura, Clifford A. Shaffer, and Eli Tilevich. 2020. Design and Evaluation of a Block-based Environment with a Data Science Context. *IEEE Transactions on Emerging Topics in Computing* 8, 1 (2020), 182–192. https://doi.org/10.1109/TETC.2017.2729585

[2] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (2017), 72–80. https://doi.org/10.1145/3015455

[3] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Evelyn Duesterwald, and Jurgen Vinju. 2009. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. 44, 10 (2009), 191–206. https://doi.org/10.1145/1639949.1640104

[4] Fei Chen, Philipp Slusallek, Martin Muller, and Tim Dahmen. 2022. Chaldene: Towards Visual Programming Image Processing in Jupyter Notebooks. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–3. https://doi.org/10.1109/VL/HCC53370.2022.9832910

[5] Thomas Cleenewerck and Ivan Kurtev. 2007. Separation of Concerns in Translational Semantics for DSLs in Model Engineering. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea) *(SAC '07)*. Association for Computing Machinery, New York, NY, USA, 985–992. https://doi.org/10.1145/1244002.1244218

[6] Martin Cápay and Nika Klimová. 2019. Engage Your Students via Physical Computing!. In *2019 IEEE Global Engineering Education Conference (EDUCON)*. 1216–1223. https://doi.org/10.1109/EDUCON.2019.8725101

[7] Söderberg Emma and Hedin Görel. 2011. Building Semantic Editors Using JastAdd: Tool Demonstration. (2011), 6 pages. https://doi.org/10.1145/1988783.1988794

[8] S. Erdweg, T. v. d. Storm, M. Volter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. v. d. Vlist, G. Wachsmuth, and J. v. d. Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[9] Martin Fowler. 2015. Language Workbenches: The Killer-App for Domain Specific Languages? https://bit.ly/32YuhJT. [Online, accessed 11 August 2021].

[10] Google. 2020. Blockly. https://developers.google.com/blockly. https://developers.google.com/blockly [Online, accessed 25 July 2022].

[11] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. ACM, New York, NY, USA, Article 270, 12 pages. https://doi.org/10.1145/3290605.3300500

[12] Michael Homer and James Noble. 2014. Combining Tiled and Textual Views of Code. In *2014 Second IEEE Working Conference on Software Visualization*. 1–10. https://doi.org/10.1109/VISSOFT.2014.11

[13] Heering Jan and Klint Paul. 2000. Semantics of Programming Languages: A Tool-oriented Approach. *SIGPLAN Not.* 35, 3 (2000), 39–48. https://doi.org/10.1145/351159.351173

[14] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. 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* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

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

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90.

[17] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo. 2020. 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)*. 1–11. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[18] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. *End-User Development: An Emerging Paradigm.* Springer Netherlands, 1–8. https://doi.org/10.1007/1-4020-5386-X_1

[19] M. Verano Merino, L. Thomas van Binsbergen, and M. Seraj. 2022. Making the Invisible Visible in Computational Notebooks. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–3. https://doi.org/10.1109/VL/HCC53370.2022.9833148

[20] Mauricio Verano Merino, Jurgen J. Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *CoRR* abs/2002.06180 (2020). arXiv:2002.06180 https://arxiv.org/abs/2002.06180

[21] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM Computing Surveys (CSUR)* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

[22] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer.* Springer London. https://doi.org/10.1007/978-1-84628-692-6

[23] Klint Paul. 1993. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (1993), 176–201. https://doi.org/10.1145/151257.151260

[24] Rascal. 2017. Pico. https://tutor.rascal-mpl.org/Recipes/Recipes.html#/Recipes/Languages/Pico/Pico.html. https://tutor.rascal-mpl.org/Recipes/Recipes.html#/Recipes/Languages/Pico/Pico.html [Online, accessed 1 September 2022].

[25] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. (1984), 42–48. https://doi.org/10.1145/800020.808247

[26] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[27] Daniel J. Rough and Aaron Quigley. 2020. End-User Development of Experience Sampling Smartphone Apps -Recommendations and Requirements. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 2 (June 2020), 1–19. https://doi.org/10.1145/3397307

[28] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. ACM, New York, NY, USA, Article 32, 12 pages. https://doi.org/10.1145/3173574.3173606

[29] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[30] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. 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* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 84–100. https://doi.org/10.1145/3426428.3426917

[31] Mark G.J. van den Brand, Arie van Deursen, Jan Heering, Hayco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3–8. https://doi.org/10.1016/S1571-0661(04)80917-4 LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).

[32] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (2000), 26–36. https://doi.org/10.1145/352029.352035

[33] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/3486608.3486908

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

[35] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2018. Bacatá: A Language Parametric Notebook Generator (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. ACM, New York, NY, USA, 210–214. https://doi.org/10.1145/3276604.3276981

[36] Dandong Yin, Yan Liu, Anand Padmanabhan, Jeff Terstriep, Johnathan Rush, and Shaowen Wang. 2017. A CyberGIS-Jupyter Framework for Geospatial Analytics at Scale. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (New Orleans, LA, USA) *(PEARC17)*. ACM, New York, NY, USA, Article 18, 8 pages. https://doi.org/10.1145/3093338.3093378

Received 2022-09-01; accepted 2022-10-02