A Cognitive Load Perspective on the Design of Blocks Languages for Data Science

Andrew M. Olney
Institute for Intelligent Systems
University of Memphis
Memphis, TN, USA
aolney@memphis.edu

Scott D. Fleming
Department of Computer Science
University of Memphis
Memphis, TN, USA
Scott.Fleming@memphis.edu

***Abstract*—The difficulty of learning data science is believed to arise from its deep prerequisites in statistics, programming, and machine learning. This poster explores how blocks languages may be used to reduce the cognitive load of learning data science. Unlike blocks languages for introductory programming, blocks languages for data science naturally align with a high level of abstraction and almost exclusively sequential execution. However, these gains in simplicity are offset by the high level of parameterization at the block level. Three designs for blocks languages are presented and compared, and implications for abstraction, sequential execution, and parameterization on cognitive load are discussed.**

***Index Terms*—blocks language, data science, abstraction, cognitive load, R**

I. Introduction

Research over the past decade has increasingly supported the positive cognitive and motivational effects of blocks languages for learning introductory programming [1]–[4]. Recent work on blocks languages has invoked data science, not as a learning domain but rather as a motivation to learn introductory programming [5], [6]. Given the positive learning outcomes of blocks languages on introductory programming, we propose exploring these effects in the context of learning data science. In this poster, we present high-level design issues for such a blocks language, focusing on the statistical programming language R and Blockly [7], [8], a configurable open-source library for creating new blocks languages.

II. General vs Data Science Programming

General-purpose programming languages typically define a small set of textual primitives that can be combined in complex ways for general-purpose problem solving. As a result, both general-purpose programming languages and their blocks language counterparts have the common control structures of sequence, iteration, and alternation (cf. selection or choice) to allow complex combinations of primitives for problem solving.

Data science programming, in contrast, focuses on data science tasks. This is not to say that languages used for data science, like R, are not general-purpose languages, but rather that languages like R are not used in a general-purpose manner when applied to typical data science tasks. Primary tasks in data science programming include obtaining data, inspecting data, transforming data, summarizing data, building models, and evaluating models [9]–[11]. These tasks not only have an internal sequential structure, but these tasks themselves are typically ordered sequentially in this way. As a result, data science programming is well suited to development environments with a linear format like Jupyter notebooks.

Because data science programming focuses on data science tasks, most programming is done at a high level of abstraction, i.e., uses libraries extensively. For example, in R, reading data from a file can be accomplished with `read.table` rather than setting up a file stream and reading loop. Similar high-level calls are used to summarize data, plot data, or build statistical models. This high level of abstraction is largely responsible for the lack of iteration and alternation structures — these control structures are hidden behind the abstraction. However, the abstraction comes at a price, because fine control must now be implemented by a high degree of parameterization rather than by complex combinations of primitives. For example, `read.table` has 25 possible arguments and returns a complex object (a `data.frame`).

III. Implications for Cognitive Load

Cognitive load theory predicts that excessive demands on working memory will impede learning [12]. In the general-purpose programming case, blocks languages may reduce cognitive load by eliminating syntactic errors (e.g., tabs, semicolons, and closing braces) and only requiring students to recognize useful blocks rather than the more difficult task of recalling code, cf. [13]. With general purpose blocks, students still have high working memory demands to plan through the complex combinations of primitives needed to achieve their goals. In contrast, data science programming, by virtue of its high level of abstraction, typically requires students to recognize a single block that accomplishes their goal, e.g., `read.table`, and add that block to their workflow pipeline. However, students must then understand how to parameterize that block, i.e., what parameters are meaningful for their use case and what the alternatives for those parameters are.

We argue that parameterization represents an unavoidable type of cognitive load that comes with a high level of abstraction, but we further argue that this particular cognitive load may actually be conducive to learning data science.

978-1-7281-4849-6/19/$31.00 ©2019 IEEE

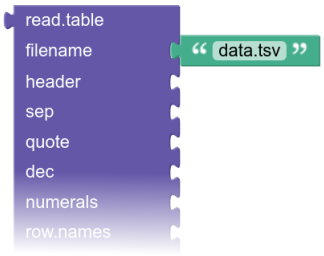

Fig. 1. The naive design for `read.table`. Only a quarter of the optional arguments are shown due to space limitations.

Cognitive load theory distinguishes between intrinsic, or necessary cognitive load, extraneous, or avoidable cognitive load, and germane, or schema-inducing cognitive load. Cognitive load is germane and positive to the extent that it creates or strengthens a schema for future problem solving. Because parameterization in high-level abstractions necessarily marks the underlying dimensions along which a class of problems typically varies (analogous to domain structure, cf. [14]), repeated exposure to parameterizations may help students learn options for problems beyond the current problem.

IV. Using Blockly for Data Science Programming to Promote Germane Cognitive Load

Given the differences between data science programming and general-purpose programming discussed above, how to design data science blocks so that their parameterizations represent germane cognitive load instead of extraneous cognitive load is an open question. We present three designs that address our previous example, `read.table`, which is a high-level R API for loading data that has 25 arguments with 24 of those being optional (`pandas.read_table` in Python is an equivalent example) and consider the cognitive load implications of each. All designs are implemented in Blockly.

A. Naive Design

The naive design in Figure 1 has a single block that enumerates all parameters and introduces extraneous cognitive load in several ways. First, by presenting all options simultaneously and without distinguishing importance, it forces the user to hold multiple options in working memory and engage in a search through the option space. Second, it does not communicate the default value for each option, even though the default may be correct for a particular use.

B. Mutator Dropdown Design

The mutator dropdown design in Figure 2 addresses the first limitation of the naive design by displaying only the obligatory argument `filename` and hiding the option arguments behind a *mutator bubble*. A mutator bubble is a Blockly convention for variable argument lists that opens a pop-up configuration window to allow the user to reconfigure a block, which keeps complexity hidden until needed. Each optional argument in this design has a fixed dropdown for its name, allowing the user to get exposure to the various options without forcing them to select settings for each option. This design should reduce extraneous load by hiding options and encouraging the user to address a single option at a time when they are added. However, the user must still recall acceptable parameters.

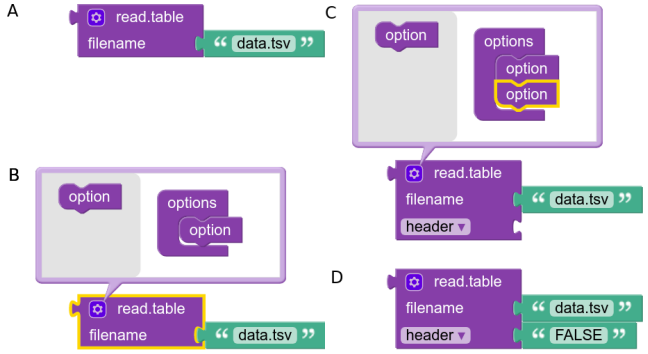

Fig. 2. The mutator dropdown design for `read.table`. The initial block (A) expands to allow addition of options (B), which appear as a dropdown in the initial block (C). Added options may be parameterized to yield the final block (D).

Fig. 3. The mutator dropdown default design for `read.table`. As in Figure 2, the initial block (A) expands to allow addition of options to yield the final block (D). Default arguments for options are automatically instantiated.

C. Mutator Dropdown Default Design

The mutator dropdown default design in Figure 3 addresses both the first and second limitations of the naive design. Like the mutator dropdown design, it hides option arguments behind a mutator bubble. However, when options are added or their values are changed in the dropdown, this design updates the parameter value with the default option. By providing default values, this design reduces the cognitive load of determining a value and also introduces germane cognitive load by fostering a schema that associates each option with its default value. Alternatively, Blockly shadow blocks can be used for setting defaults.

V. Conclusion

Blocks languages used for learning introductory programming hold promise for learning data science. However, our preliminary design work suggests that a naive implementation of blocks languages for data science will create extraneous cognitive load. By incorporating design elements that hide API complexity until needed and then provide default suggestions for using the API, we believe that we can reduce extraneous cognitive load and foster germane (schema-inducing) cognitive load that will generalize learning to novel problems. If these hypotheses are correct, then learning data science programming with appropriately designed blocks languages may be substantially easier than learning general-purpose programming with blocks languages. The major limitation of our work is that it focuses on functions rather than objects (e.g., `data.frame`). Incorporating objects into data science blocks languages is a subject for future work.

References

[1] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From Scratch to "real" programming," *ACM Transactions on Computing Education*, vol. 14, no. 4, pp. 25:1–25:15, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2677087

[2] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper,"Mediated transfer: Alice 3 to Java," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, pp. 141–146, 2012. [Online]. Available: http:/doi.acm.org/10.1145/2157136.2157180

[3] B. Moskal, D. Lurie, and S. Cooper, "Evaluating the effectiveness of a new instructional approach," *ACM SIGCSE Bulletin,* vol. 36, no. 1, pp. 75–79, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1028174.971328

[4] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. "Scratch: Programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592761.159

[5] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura, "BlockPy: An open access data-science environment for introductory programmers," *Computer*, vol. 50, no. 5, pp. 18–26, May 2017.

[6] A. C. Bart, J. Tibau, D. Kafura, C. A. Shaffer, and E. Tilevich, "Design and evaluation of a block-based environment with a data science context," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–12, 2018.

[7] Google, "Blockly," 2019, original-date: 2013-10-25T21:13:33Z. [Online]. Available: https://github.com/google/blockly

[8] 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 '17)*, pp. 21–24, Oct. 2017.

[9] C. Byrne, *Development Workflows for Data Scientists*. O'Reilly Media, Inc., 2017.

[10] R. Schutt and C. O'Neil, *Doing Data Science: Straight Talk from the Frontline*. O'Reilly Media, Inc., 2013.

[11] A. F. Zuur, E. N. Ieno, and C. S. Elphick, "A protocol for data exploration to avoid common statistical problems," *Methods in Ecology and Evolution*, vol. 1, no. 1, pp. 3–14, 2010. [Online]. Available: https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/j.2041-210X.2009.00001.x

[12] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive Load Theory*. Springer, 2011.

[13] E. Tulving, "How many memory systems are there?" *American Psychologist*, vol. 40, no. 4, pp. 385–398, 1985.

[14] D. Ratiu, M. Feilkas, and J. Jurjens, "Extracting domain ontologies from domain specific APIs," in *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR '08)*, pp. 203–212, IEEE Computer Society, 2008. [Online]. Available: https://doi.org/10.1109/CSMR.2008.4493315