Guest Editor Introduction to the
VLSS Special Issue on Blocks Programming

Franklyn Turbak
Computer Science Department
Wellesley College, Wellesley MA, USA
fturbak@wellesley.edu

This special issue of the *Journal of Visual Languages and Sentient Systems (VLSS)* focuses on *Blocks Programming*. Over the past twenty years, blocks programming languages have evolved from research lab prototypes to practical tools used by tens of millions of people. The core idea behind these languages is simple: rather than constructing programs out of sequence of characters that are lexed into tokens that are parsed into syntax trees, why not construct syntax trees more directly by composing drag-and-drop visual fragments that represent the nodes of the trees?

There are now a host of blocks languages that can be used for a wide variety of purposes, such as creating games, scripting 3D animations, inventing mobile apps, developing agent-based simulations, controlling robots, generating 3D models. and collecting/analyzing/visualizing scientific data.

Educators have seized upon blocks languages as a way to introduce beginners to computer science concepts. Environments like Scratch, Blockly, Pencil Code, App Inventor, Snap!, AgentSheets/AgentCubes, and Alice/Looking Glass have become popular ways to introduce students (from grade schoolers through graduate students) to computational thinking and programming. This is done not only in traditional classrooms, but also in extracurricular settings via online activities like Code.org's *Hour of Code*.

But blocks environments are also a boon to a broader population whose primary goal is to "make stuff" rather than learn computer science principles. This includes artists, scientists, hobbyists, entrepreneurs, and other end users and so-called *casual programmers* who want to build computational artifacts for themselves, their families, their workplaces, their communities, and the world at large.

There are many questions surrounding blocks languages. Why are they so popular? What are their key advantages and limitations compared to other programming environments? Do they really lower barriers to programming, or is that just hype? In what ways do they help and hinder those who use them as a stepping stone to traditional text-based languages? What are effective pedagogical strategies to use with blocks languages, both in traditional classroom settings and in informal and open-ended learning environments? How does the two-dimensional nature of blocks programming workspaces affect the way people create, modify, navigate, and search through their code? Do blocks environments have any features that are worth incorporating into IDEs for traditional programming environments? What are effective mechanisms for multiple people to collaborate on a single blocks program when they are co-located or are working together remotely? Can blocks-based environments be made accessible to users with visual or motor impairments?

These questions are being investigated by educators, software developers, and researchers from areas that include visual languages, human-computer interaction, programming languages, education, psychology, learning science, and data analytics. This special issue gives a sample of the kinds of research and development that is taking place in the blocks programming community.

Many blocks languages are *domain-specific languages (DSLs)* that are not general programming systems but are instead targeted at tasks in a particular domain. An excellent example of such a DSL is *SparqlBlocks*, a blocks-based environment for making queries to find information encoded in linked data within the semantic web. This system is described by its creators, **Miguel Ceriani and Paolo Bottoni**, in the first article of this issue, *SparqlBlocks: Using Blocks to Design Structured Linked Data Queries*. The *SparqlBlocks* programming environment highlights several key advantages that blocks languages have with respect to traditional textual notations (in this case, the text-based SPARQL query language). First, *SparqlBlocks* language features are represented by drag-and-drop blocks whose shapes both suggest how they can be composed and only allow syntactically valid connections to be made. Second, generating and editing queries involves manipulating high-level chunks of information (blocks) as opposed to low-level character representations, which not only aids the cognitive task of assembling queries but also avoids frustrating errors due to misspelled keywords or incorrect punctuation. Third, the blocks of the language are organized into categories within a toolbox menu, allowing users to learn available functionality by exploring the menu options and use recognition rather than recall to find the constructs that that they need. Fourth, the system has a *liveness* property such that query results are automatically displayed as soon as a valid query is created and updated as soon as the query is modified. (Although liveness has nothing to do with blocks per se, it is a common feature of many blocks environments.)

In addition to these standard benefits, *SparqlBlocks* has an innovative twist: the query result tables themselves are represented as blocks, which encourages the use of result fragments in new queries. Anyone who has programmed using JSON data representations or Lisp-like s-expressions will appreciate the power unleashed by having output data representations that can easily be repurposed as input data within a program.

SparqlBlocks is clearly not a language targeted at children, but at people who understand the Resource Description Framework (RDF) framework of the semantic web. So it serves as an interesting case study for those who are dismissive about blocks languages, and claim they can't be used for "real" coding or are just a stepping stone for programming in traditional text-based languages. In this case, it's easy to imagine a class of users who would be completely satisfied with using *SparqlBlocks* for all their semantic web queries without ever feeling the need to learn the textual SPARQL syntax. Perhaps the naysayers are confused by the brightly colored blocks and the fact that many current blocks languages have limited environments that are targeted at those who have never programmed before, and so they incorrectly conclude that such environments are only good for kids and other newbies and could never be of use to adults or seasoned programmers. But fundamentally, blocks languages are just an alternative way of representing the syntax trees that underly any program, so there's no reason that they can't express everything that a traditional text-based language can express. And it's worth remembering that it was not so long ago that "real programming" was done only in assembly code and higher-level programming languages were such a futuristic idea that their study was considered a branch of artificial intelligence. It will be interesting to see what is considered "real programming" twenty to forty years hence.

This is not to say that blocks languages don't have downsides. Common complaints are that blocks representations are are less dense than text representations (so less information can be conveyed in one screenful) and that they have high *viscosity* — creating, moving, connecting, disconnecting, copying, and deleting blocks when editing a program can take more operations and time than entering and editing text. And syntactically correct programs aren't necessarily semantically meaningful. For example, in *SparqlBlocks*, many users struggle with the correct usage of literals and variables in their queries.

An important line of research in blocks based languages is mitigating these downsides. The next three articles in this special issue address this theme.

One idea for combining the benefits of blocks and text within a single environment is *bidirectional mode switching*, in which there is a way to convert a blocks program workspace into an editor buffer with an isomorphic text program and vice versa. There are now several blocks programming environments that support bidirectional mode switching, but one of the earliest was Tiled Grace, which is described by its developers **Michael Homer and James Noble** in their article *Lessons in Combining Block-based and Textual Programming*. Tiled Grace is a blocks version of the existing Grace text-based language. Users can edit a Tiled Grace program either in blocks mode or text mode and, with the press of a button, see their program transformed by a smooth animation into the other mode. Each mode has its own advantages. Blocks mode allows editing the program syntax tree with meaningful chunks, provides a toolbox that organizes the language constructs into inspectable categories, and provides visual feedback for syntax errors and program dependencies (e.g., associating references of variable and method names with their declarations). Text mode allows low-viscosity editing of a denser representation of the same program.

The article describes an experiment in which early undergraduate Java programmers were asked to write, modify, correct, and describe Tiled Grace programs, and were allowed to switch between blocks mode and text mode as many times as they wanted to in order to complete their tasks. One interesting result is that participants switched modes numerous times in performing the tasks. Many participants (who were already used to text-based programming in Java) found drag-and-drop editing with blocks frustrating, so would switch to text mode to edit the programs. But some participants would switch to blocks mode to perform certain kinds of edits. Intriguingly, over a third of switches to blocks mode involved no edits, implying that participants were using it just to examine the structure of the code.

The authors compare their study to some other studies involving bidirectional mode switching. These other studies involved different environments, different tasks, and participants with different programming backgrounds, so it's not surprising that there is a lot of variability between results. But clearly, bidirectional mode switching is a rich area for future study. and can give insight into the relative advantages of blocks vs. text for creating, modifying, debugging, and understanding programs.

A different approach to mediating between blocks and text is described in the *Frame-Based Editing* article by **Michael Kölling, Neil C. C. Brown, and Amjad Altadmri**. They describe a novel editing paradigm based on *frames* that is the basis for an editor for the Java-like Stride language that has been incorporated into their Greenfoot and BlueJ programming environments. Rather than switching between blocks and text views, the frame-based editor combines key aspects of both blocks and text in a single unified view.

Declarations and statements are represented as frames, which are visual block-like entities that can be created, moved, copied, and deleted as single syntactic units. Frames have slots for nested program structure. There are two types of slots. A frame slot is filled with another frame, while a text slot is filled by typing characters on the keyboard (or copying and pasting text from elsewhere). Text slots are used for entering expressions (e.g., numbers, variables references, arithmetic/relational/logical operations, etc.) and specifying things like parameter names, types, and the assigned variable in an assignment statement. Although frames and frame/slot cursors

can be selected/dragged/positioned with a mouse, there are keystrokes for creating, selecting, copying, pasting, and deleting frames and for navigating to frames and slots. This makes it possible to edit Stride programs using only keystrokes, even though frames are block-like features.

The Stride frame-based editor supports numerous other features that give it a low-viscosity "feel" more like a professional text-based Integrated Development Environment (IDE) than a high-viscosity visual language. The article describes a few usability experiments with the Stride editor, some of which are positive but others of which indicate areas for improvement. More detailed experiments with frame-based editing and comparing it with editing in other blocks based environments are obvious areas for future work, as are incorporating frame-based features into other blocks-based environments and adapting frame-based editors to other text-based languages.

This article also describes how frame-based and blocks-based editing arose out of two research threads: one involving structure editors for text-based programs, and another involving visual programming languages. Members of the blocks community should familiarize themselves with both of these lines of research.

Alex Repenning's article *Moving Beyond Syntax: Lessons from 20 Years of Blocks Programming in AgentSheets* is his personal reflection about the history and future of blocks programming. Repenning's 1995 AgentSheets environment for rule-based programming of agents in a two-dimensional grid included a blocks programming editor for specifying the rules that control the agents. Repenning defines four affordances for blocks programming that he uses as a lens to discuss the history of blocks programming and related work in visual programming. Readers interested in the historical underpinnings of blocks programming should follow up on the work cited by Repenning as well as the structure editor work summarized by **Kölling, Brown, and Altadmri**.

The main thrust of Repenning's article is that although blocks languages have clear benefits for making the syntax of programming less frustrating for beginners, developers of these environments should focus more on improving their semantics and pragmatics (which he defines as *the study of what code means in particular situations*). He gives some compelling examples from AgentSheets and the follow-on AgentCubes languages: automatically generated context-based documentation and error messages, and so-called *conversational programming* and *live palette* techniques that support context-based debugging and prediction of program behavior — e.g., dynamically showing the values of rule conditions and which rules would fire as agents are dragged through different cells in a grid world. Such feedback can give programmers a better understanding of the *behavior* of their programs, not just their *structure*.

Repenning also argues that environments for beginners need to focus less on traditional programming skills and more on computational thinking and problem-solving skills. In this context, there are advantages to having restricted computational thinking environments that are tuned to having relatively simple solutions to certain classes of problems rather than general programming tools that can solve any problem. Repenning notes that solutions with general programming tools are often swamped by *accidental complexity*, whereas in more constrained environments, it is easier to support pragmatics. Many of the semantic, pragmatic, and computational thinking benefits mentioned by Repenning are strongly rooted in the particular grid-with-agents-controlled-by-rules paradigm embodied in his AgentSheets and AgentCubes environments. But how can these sorts of benefits be realized in other paradigms, domains, and modalities? This is a rich area for future research. As a concrete challenge to the reader, I suggest considering how to apply some of Repenning's ideas to **Ceriani and Bottoni's** *SparqlBlocks* system, or some other blocks language of their choice.

Although Repenning defines four key affordances of blocks programming, he doesn't explore in detail *why* those affordances should help blocks programmers. In their article *How Block-based Languages Support Novices*, **David Weintrop and Uri Wilensky** propose a framework for understanding how such affordances help learners construct meaning from the environment in the context of solving a particular problem. They use a notion of *webbing* to describe all resources available to the learner in a situated computational problem solving task. Their framework involves two dimensions: the first dimension distinguishes internal cognitive processes vs. external communication; the second dimension distinguishes generating a program vs. interpreting one. Based on the 2 x 2 matrix involving these dimensions, they describe four roles for the learner. They then give examples from a study involving RoboBuilder (a blocks-based environment for describing the behavior of screen-based robots that battle each other) of how affordances of blocks help learners in each of the four roles. An interesting aspect of their analysis is it suggests that designers should consider all four roles when designing a blocks language. However, there may be trade-offs in design that benefit one role at the expense of another. Elaborating on design decisions and trade-offs implied by this framework, as well as applying it to non-blocks programming languages, are avenues for future research.

Blocks programming environments are used in many situations where novice programmers make heavy use of example programs to learn features of the language, often with little or no help from an instructor or peer. In order to design helpful examples and incorporate them into the learning process, it is essential to better understand how blocks programmers learn via examples. This is the motivation behind the work by **Michelle Ichinco, Kyle Harms, and Caitlin Kelleher** in their article *Towards Understanding Successful Novice Example Use in Blocks-based Programming*. They investigated two features of examples in Looking Glass: (1) different styles of annotations that explain how an example works and (2) surface and structural similarities between and example and the assigned task (which involved modifying a given program).

In a study of 99 children aged 10 to 15, they found that annotations on examples (vs. no annotations) helped participants complete a similar task, but not a dissimilar one. Also, subjects completed more tasks using similar examples than dissimilar ones, and the similarity was correlated with getting to a later stage in a 4-stage model of task completion. However, the tasks were challenging for the subjects, and many subjects did not get past the 1st of the 4 stages for many of the tasks. More work clearly needs to be done to help novices get unwedged from this 1st stage.

To understand the level of understanding that subjects had of the relationship between the examples and tasks, subjects were asked to draw analogical mappings that connected parts of the example program to corresponding parts of the (unmodified) task program. The study found strong correlations between annotations and correct mappings and between example similarity and correct mappings, but only a weak correlation between correct mappings and task success.

Finally the study looked at how programming behavior was correlated with task completion. For example, editing the program in the 1st stage or performing many user interface operations in the 2nd stage were strong indicators that subjects would *not* complete the task.

This study indicates that example-based learning in blocks programming may not be particularly effective. It remains to be seen whether more carefully crafted examples with more explicit analogical mappings, possibly enhanced with other forms of scaffolding, can increase the effectiveness of examples.

The special issue concludes with the topic of accessibility of blocks programming environments. Many people naturally assume that the visual, drag-and-drop nature of blocks languages inherently makes them inaccessible to those with visual or motor impairments. But this is not necessarily the case! The fact that blocks programming reifies syntax tree nodes and focuses on creating, modifying, and navigating syntax trees may be able to support programming for those with disabilities better than character-based languages.

In their research notes paper *Design Considerations to Increase Block-based Language Accessibility For Blind Programmers Via Blockly*, **Stephanie Ludi and Mary Spencer** describe their work-in-progress on extending the Blockly browser-based blocks language framework to support visually impaired users. One aspect of their work is extending Blockly with screen reader support so that it can pronounce block assemblies (including unfilled sockets) in a meaningful way. Another is providing keyboard support for navigating, connecting, disconnecting, etc. blocks and using menus on the workspace. (As discussed in Sec 14.5 of **Kölling, Brown, and Altadmri's** article, the fact that their frame-based Stride editor already has complete keyboard support for all editing operations makes it an interesting candidate for accessibility.) They also reference work on vocal interfaces that can make blocks programming accessible to people with motor impairments.

In the spirit of promoting accessibility, all researcher who develop and study blocks environments are encouraged to become familiar with ways of making these environments more accessible.

Many individuals contributed to this special issue. I am grateful Professor S-K. Chang, the Editor-in-Chief of *VLSS*, for his support and encouragement for this special issue. Many thanks to the authors who responded to the call for papers for this issue. Of the nine papers that were submitted, six were accepted as full papers (subject to revisions) and one was accepted as a research notes paper. I especially appreciate the willingness of the authors of the accepted papers to go through three rounds of revisions on their papers (two based on reviewer feedback and a final one based on my feedback), which led to very high-quality results. Finally, I am indebted to our reviewers, whose constructive comments during two rounds of reviews greatly improved the quality of the papers.

Preparing this special issue has been a rewarding experience. It has underscored how much exciting work is going on in the area of blocks programming and exposed fertile ground for future research, both in terms of studying how people use and learn from blocks environments and improving the environments to address numerous problems that have been identified in existing systems.

There is a vibrant community of researchers conducting work on blocks programming. Join us!