Invited Panel: The Future of Blocks Programming

Caitlin Kelleher
(Looking Glass)
School of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
ckelleher@wustl.edu

John Maloney
(GP)
HARC/Y Combinator Research
jmaloney@media.mit.edu

Paul Medlock-Walton
(Gameblox)
Scheller Teacher Education Program
MIT
Cambridge, MA, USA
paulmw@mit.edu

Evan Patton
(App Inventor)
MIT
Cambridge, MA, USA
ewpatton@mit.edu

Daniel Wendel
(StarLogo Nova)
Scheller Teacher Education Program
MIT
Cambridge, MA, USA
djwendel@mit.edu

***Abstract*—Five blocks programming environment developers were invited to share their thoughts about the future of blocks programming and seed a discussion among all attendees about this topic for the final session of the workshop.**

CAITLIN KELLEHER (LOOKING GLASS)

Learnable Blocks

Blocks programming environments need to provide more support for learning to program as users are working on their own programs. In recent years, we have seen blocks-based languages and environments move into a lot of new domains. This workshop alone showcases languages designed for industrial robotics, program correctness proofs, and web application prototyping. One of the important motivations behind creating blocks languages for new domains is to open programming in those domains to a wider collection of people. In many of these domains, users will attempt to learn by exploration as they attempt to create programs that solve specific problems. My group has been working for several years on trying to support independent learning in programming environments. One recent study we did found that in unconstrained situations, young novices tend to gravitate towards in-context forms of help, and particularly suggestions generated based on static code analysis. However, much of the effort that has gone into developing learning support for programming environments has focused on forms of help in which the user does not directly control their task, such as tutorials and, more recently, code puzzles. With a coming increase in the numbers and types of users exploring blocks languages and environments in order to solve personally identified problems, we think the community needs to be exploring the design of effective help for use as novices work on their own tasks.

JOHN MALONEY (GP)

Blocks all the Way Down: A Blocks Programming System Implemented in Blocks

Scratch 1.x had a little secret. An obscure gesture dropped the user through a trapdoor into a completely different programming world, the one used to implement Scratch itself. That world included not only all of the Scratch source code, but also all the programming tools used to create Scratch: code editor, debugger, object inspector, search facilities, code management tools, etc. This secret trapdoor was a great help to the team as Scratch was developed but it was never intended for use by anyone except the Scratch developers.

However, the secret leaked out. Then an interesting thing happened.

A few Scratchers discovered that they could modify Scratch, adding new blocks and even adding new features to the Scratch user interface. Those Scratchers told others, and soon a subculture of Scratch modifiers developed, with Scratchers sharing their "Scratch mods" and techniques with each other. The Scratch Wiki now lists nearly 90 Scratch mods.

The Scratch modding phenomenon is especially surprising given that Scratch 1.x was written in Smalltalk, a non-mainstream, text-based programming language with far less documentation and community support than more popular programming languages. It required great motivation and resourcefulness to dive into that world, and only a relative small fraction of Scratchers did so.

GP (gpblocks.org) asks the question: What if we designed a blocks environment that encouraged and supported modification and extension? Would that enable more users to have experiences like the Scratch modders?

The potential rewards of doing this are great. Those who choose to drop through GP's trap door can learn about data structures, such as lists and hash tables. They can explore algorithms for things like paint bucket fill, line drawing,

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

sorting, or detecting prime numbers. Most important, they can feel the empowerment that comes from being able to add their own blocks and user interface features to the system.

How does GP encourage such exploration? In GP, unlike Scratch 1.x, the entire blocks programming system is implemented in GP itself, so all the system code can be viewed as blocks. Users don't need to struggle with syntax or learn a new programming language. Of course, users who dive into GP's implementation still have many new things to learn. However, learning those things is easier because they don't need to learn a new programming language at the same time.

We hope that GP will empower more budding programmers to explore and extend the system itself. Not all GP users will be interested in that, but even those who merely dip a toe into those waters can benefit, since it quickly becomes obvious that real applications like GP are built out of code much like their own. That realization is powerful, perhaps even life-changing.

PAUL MEDLOCK-WALTON (GAMEBLOX)

Powerful Environments and Simple APIs

Many software projects provide APIs as an easy interface to complicated applications: from translation and image recognition services which mask the complexity of machine learning algorithms, to cross platform mobile libraries which abstract the differences between various mobile operating systems. While these APIs are typically aimed at developers with programming experience, blocks based programming languages provide novices with simpler ways of interacting with these powerful systems.

At the MIT Education Arcade, we are using Gameblox, a blocks based programming language for creating games, to make a set of participatory simulations. Participatory simulations enable students in a classroom to use their mobile devices to act as individual agents within a larger simulation. Building these simulations often requires developing a mobile app which supports peer to peer communication through QR codes or NFC, a server capable of managing communication between multiple devices, and a computer application to display graphs or other data displays that are updating in real time.

Instead of needing a team of engineers who can master these technologies, Gameblox allows users to simply drag and drop elements on an interface and connect blocks together in order to create participatory simulations. As Gameblox and other blocks languages move toward the future, we find that harnessing the capabilities of powerful environments, including virtual reality, hardware, and server clusters running machine learning algorithms, into simple blocks provides a way for non-experts to creatively interact with previously inaccessible technology.

EVAN PATTON (APP INVENTOR)

Abstracting the Blocks Abstraction

App Inventor has two main goals: computational thinking pedagogy and, through improved computational thinking, empowerment of people to better their communities. It is important that the blocks languages we build in some way approximate the mental models of those using the language to reduce the cognitive workload required for new users of the language while at the same time offering them sufficient power to accomplish their goals. Improving the mapping of the language to mental models can be done through new primitives for a particular problem space to decrease the amount of effort needed to solve a problem; it can be through localization of the language to make it approachable to the non-English world; it can be through new tools to visualize or map out the problem and to move away from low level language primitives that place the user too far removed from the problem at hand. Very few people in the world will feel empowered if they have to write assembly code to solve their problems, and looking forward to the future of blocks languages my expectation is that moving toward higher levels of abstraction will be of interesting research and pedagogical value. The abstractions that blocks and other visual programming languages provide will be a determining factor in their use to empower people to solve problems.

One issue that comes up often is the issue of blocks-to-text modes; the blocks-to-text issue is a double-edged sword. While on the one hand App Inventor often sees users who think they are not doing "real coding" because the language is not textual, that therefore the language is somehow inferior, blocks-to-text may be useful for pedagogical reasons as a bridge for students to understand the relationship between one's perception of coding and the act. Someday though we might look upon blocks-to-text in a manner similar as looking at the assembly generated by a compiler is useful to learn about its translation unit. There are a few reasons that this does not have to be true:

1) Localization: We can localize blocks more easily than we can localize existing text languages. App Inventor supports 11 languages for programming Android, for example, but if one wants to represent the same logic in Java it must be expressed in English-only keywords. This is antithetical to empowerment because one must first learn another language before being able to program. Blocks-to-text implementers ought to think about how to handle the localization challenge very carefully so that it is equally a great feature to all, not just English speakers.
2) Organization: The multidimensional space afforded in the 2D plane of the blocks editor allows for organizing code (and, in a way, thoughts) differently from the linear top-down text representations. Again, this might better match how some people visualize the flow of the information, and makes the code more approachable because what conventions exist are enforced by the editor in the loop rather than with a compiler as part of a write, compile, debug lifecycle. 3. Abstraction: My first experience with blocks programming was as a teenager using the programmable LEGO MINDSTORMS yellow brick. In so much as the blocks for programming the robot looked like bricks similar to the physical bricks

> I built with in the real world, there was a mapping to an existing knowledge space that was familiar. However, in the future of blocks languages there isn't any reason why we couldn't provide blocks that look like drone parts, robot parts, or pictures of different tools needed to make a recipe. Abstracting to different representations to better match the primitives of the task at hand is an extremely powerful way of helping empower others to use computational thinking to their advantage.

In summary, blocks potentially are a powerful tool to get people who would not normally be inclined to code because it is "too hard" to actually do some coding. The real benefit, however, is the fact that blocks languages could be better used for tasks like modeling domain-specific languages that fit into one's problem space. We have seen a general progression from lower level languages closely matching machine architectures and languages toward higher level languages better matching how humans represent and solve problems. Blocks languages can be another step in this progression as we build languages intended for anyone to solve problems, and by doing so make the language even more approachable, especially when coupled with appropriate levels of abstraction, organization, and localization. This is an advantage over existing textual languages and one that is still ripe for research.

DANIEL WENDEL (STARLOGO NOVA)

Ideal Representations for In-School Integration

Several attempts have been made to optimize beginner-friendly editing environments for certain input modalities or user constraints; for example, editing with an Xbox controller, editing on a tablet screen, editing by non-readers, or editing without certain types of visual cues (e.g. for users with limited vision or colorblindness). These are interesting directions for continued research. But how do the resulting tools compare to tools for use with a typical laptop computer, by users with average vision and manual dexterity? Moreover, how do those "typical" environments compare with each other, with their differing metaphors and design paradigms? While some early work has been done in identifying and cataloging design features that are well-suited for different contexts, I have not seen many studies that would help us to agree upon patterns as "best practices". Will we get to the point of abandoning early but widely-used designs in favor of designs with firmer grounding in design research and experimental data?

One specific area the StarLogo Nova team is interested in is custom blocks and/or visual representations for specific science subject areas. Several projects have attempted general-purpose computing with blocks, including PyBlocks, ArduBlocks, PencilCode, and GP, and other tools like Scratch and App Inventor are specific to a particular output format but aim to fully expose the capabilities within that space, following Seymour Papert's "low floor, high ceiling" philosophy, with Mitchel Resnick's "wide walls" addendum. The StarLogo Nova team is just beginning a project with Doug Clark (Vanderbilt/Calgary) and Okhee Lee (NYU) that will explore a twist on that philosophy: by carefully limiting flexibility ("lowering the ceiling" or "narrowing the walls"), can we provide genuinely meaningful and engaging computational model-building experiences, within specific content areas, for students with even less ramp-up time? Furthermore, if we can do this independently for multiple content areas (e.g. ecosystems and Newtonian mechanics), are there gains across them? Or does the lack of generalization of the underlying modeling platform lead to a lack of generalized understanding on the part of the students?

A separate project along a similar vein is the Linking Complex Systems project with the Carolyn Staudt and Chad Dorsey at The Concord Consortium, which asks, "How can agent-based models and systems dynamics models be linked or used in tandem to provide students with deeper insights into complex problems?" This project explores fundamental questions about what an "optimal" representation is for a particular concept, or whether multiple representations are needed. Whereas StarLogo Nova's direction previously was to attempt to represent all modeling concepts as imperative programming concepts through novel blocks-based designs, we are now beginning to imagine declarative or even higher-level representations that indicate relationships rather than specific actions. In this way, the conversation in StarLogo Nova has moved away from "how do we visually represent this programming concept?" to "how do we visually represent this computational modeling concept?".

I believe that the fundamental questions of "ideal representation" and of general vs. special-purpose computing will lead to continued breakthroughs in the democratization of computing and in computing education over the next decade.