Lambda in Blocks Languages: Lessons Learned

Brian Harvey
Computer Science Division
University of California
Berkeley, CA, USA
bh@cs.berkeley.edu

Jens Mönig
SAP SE
Göttingen, Germany
jens@moenig.org

***Abstract*—In designing BYOB and Snap*!*, we wanted to extend the Scratch idea of visual metaphors for control structures to include anonymous procedures and higher order functions. We describe the iterations in the design leading to the current "grey ring" notation.**

Keywords—blocks language, BYOB, Snap!, visual metaphor, control structure, lambda, higher order functions.

I. Introduction: Visual Metaphors

Scratch wasn't the first drag-and-drop programming language, but it pioneered the use of carefully designed visual metaphors for computer science ideas, such as snap-together Lego-brick-esque blocks (procedures) to form a script (thread), the C-shaped block for a control structure that wraps around a script, and the subtle upward arrow on those control blocks that implement looping (Figure 1).

In 2010 we collaborated on a project to extend the power of Scratch to more advanced computer science ideas, suitable for an undergraduate CS course. The first of these was the ability to create new blocks by, essentially, giving a script a name. This capability was important not only for the sake of modularity — Scratch kids would write scripts hundreds of lines long that no adult could read — but also because defining procedures opens the door to recursion, one of the central CS ideas. This first extension (done by Jens before Brian got involved) was therefore named Build Your Own Blocks, or BYOB.

In representing the script that defines a new block, we wanted to follow the Scratch principle of visual metaphors. Scratch has hat blocks that are used to specify an event that should trigger the running of a script (Figure 2).

In defining custom blocks, we wanted to convey the idea "this script should run when the new block is used," so we put

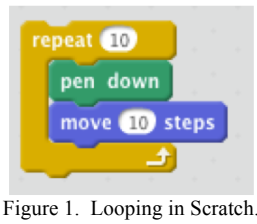

Figure 1. Looping in Scratch.

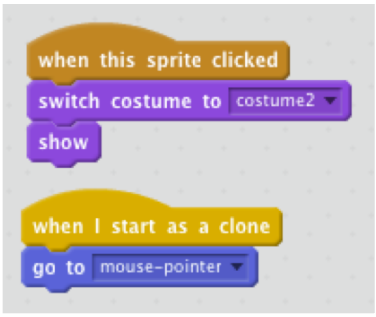

Figure 2. Event blocks in Scratch.

the block's picture in a hat block above its script (Figure 3). (The BYOB block shapes were based on those of Scratch 1.4, so they look a little different from the new Scratch 2.0 blocks.)

Scratch users already understand that a hat block means "when this, do that"; given that understanding, the meaning of the hat block in the script that defines a custom block is clear.

II. Anonymous Procedures

For the new introductory CS course then being developed at Berkeley, we weren't satisfied with recursion. We also wanted to teach another central CS idea: higher order functions — functions that take functions as arguments. To make this work, we needed to be able to distinguish between invoking a block (the usual thing in Scratch) and using the block itself as a value (e.g., as input to a higher order function). In CS terms, we needed to be able to encapsulate an expression as a procedure; that is, we needed anonymous procedures. Saying that another way, we needed the Lisp λ. For the introductory students, our emphasis is on making it easy to use higher order functions; at the same time, we want experts to be able to provide students with arbitrary control structures in libraries, and so we want to provide the full power of lexically scoped λ.

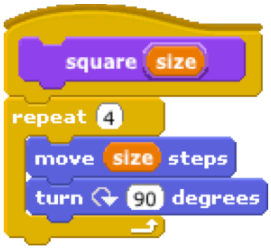

Figure 3. Custom blocks in BYOB.

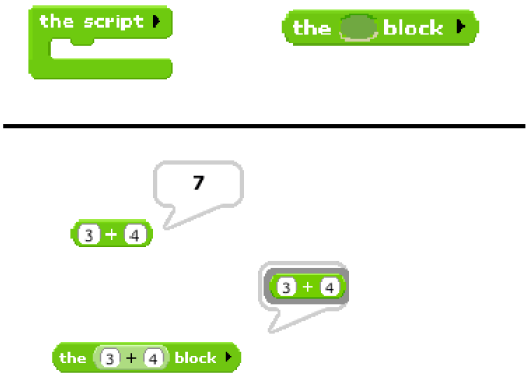

Figure 4. Initial λ representation in BYOB.

A. *Anonymous Procedures in BYOB*

In a blocks language, the obvious way to introduce a new mechanism is as a block. Two blocks, actually, because we needed one for commands/scripts and another for reporters/expressions (Figure 4.) The name `the block` dates from early in the design process, when we weren't sure whether a single command block should go in `the script`, as suggested by its shape, or in `the block`, as suggested by its name. The final decision was that `the block` accepts only reporters, so perhaps we should have renamed it "the reporter" or "the expression."

As shown in the example, even at this stage the visual metaphor of a block in a speech balloon is very powerful in conveying the idea of procedure as data. We drew a grey border around such blocks and scripts to further emphasize that the procedure was not being invoked in the current context.

Technically, in a lexically scoped language a procedure is not the same as the expression it encapsulates, because it also encapsulates an *environmant* that includes local variable bindings of the context in which this procedure was defined. In Scheme, the language we used as our model in this work, there is no standard output representation of a procedure, not even as text, let alone a visual metaphor. The grey border around an anonymous procedure is also a sort of fingers-crossed indication that there's more to a procedure than meets the eye. We try not to tell lies in the pictures we draw, but we don't always tell the entire story. This is true even in Scratch; consider the looping blocks in the two scripts in Figure 5. In the script on the left, the `repeat` block evaluates the variable `count` just once, before the loop begins, so the loop runs four

Figure 5. Special forms in Scratch.

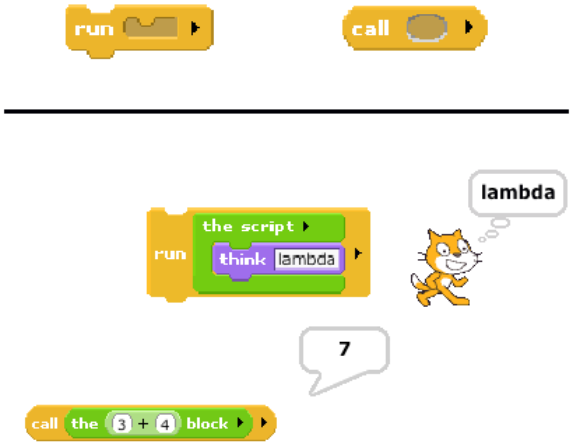

Figure 6. Applying an anonymous procedure.

times even though the value of `count` is 8 by the time the loop finishes. By contrast, the otherwise very similar `repeat until` block re-evaluates the `>` test each time through the loop. In Scheme terms, `repeat until` is a special form, whose Boolean argument is evaluated in normal order rather than the usual applicative order. (Actually, all C-shaped blocks are special forms, because their script argument isn't pre-evaluated.) Nothing in the appearance of the two blocks provides a metaphor for that difference

The point of encapsulating a procedure is that eventually it will be invoked. Since anonymous procedures may be inside another procedure, they don't have blocks in the global *palette* (the menu of blocks to the left of the scripting area). Instead we need generic blocks that take the desired procedure as an argument (Figure 6).

B. *Arguments to Anonymous Procedures*

So far, our examples have been of procedures with no arguments. But procedures are much more powerful when they embody a family of actions, controlled by argument values when they are invoked. Our lambda blocks can be expanded, as in Figure 7, to include formal parameters, which can be dragged into the expression being encapsulated. The `call` block can similarly be expanded to provide actual arguments.

At this point, the core intellectual work was done: We could create and invoke anonymous procedures with inputs, and use that capability to write higher order functions.

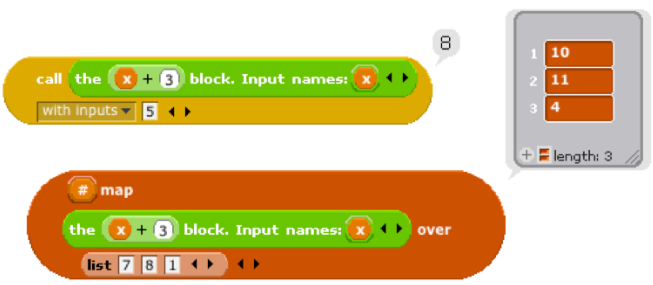

Figure 7, Explicit formal parameters.

Figure 8. `Call` block with grey ring input.

C. *Efforts to Improve the User Interface*

The lambda implementation as described so far had both pragmatic and pedagogic problems. Starting with the pragmatic, it was just too much trouble always to be searching for the lambda blocks at the bottom of the Operators palette. Also, especially for reporters, expressions using λ ended up being too wide to fit in the scripting area. Each of us proposed partial solutions. First, shift-clicking on any block or script displayed a one-item menu, namely "quote," which if clicked would surround the block or script with the appropriate λ block. This solution had the defect of not being something a user would automatically discover, but it was very convenient in avoiding the need to pull a block from Operators. But it didn't solve the width problem. The other solution was to take advantage of the fact that input (argument) slots in a block knew if they were expecting a procedure input. The idea was to reuse the grey rings around encapsulated procedures when displaying them as an *input* notation also. We changed the procedure-type input slots from being the same color as the block to being grey, and they could put a grey ring around any procedure dropped into them (Figure 8). This new notation was a dramatic improvement in the width problem. (You'll notice that the formal parameter is missing. This point is coming up in the next paragraph.) The trouble is that *sometimes* a user wanted to call a function that would provide the function input to the `call` block, and so wouldn't want that call encapsulated (Figure 9). We were therefore forced into what turned out to be a very difficult part of the interface for users to understand: An expression dropped into a procedure-type input slot would be ringed or not ringed depending on how close to the slot the expression was dropped (Figure 10). In retrospect this is the only truly embarrassingly bad design decision we made in BYOB

About formal parameters: The connection between formal parameters and actual arguments is always difficult for beginning programmers. Brian's mom was a math teacher during the post-Sputnik New Math days, and he remembered one of the lessons of those curricula: If you show an eight-year-old an equation such as $x+3=7$ and ask what x is, you're unlikely to get an answer. But if you show the same kid $\square+3=7$ and ask "what number goes in the box," s/he can solve it easily. We allow empty input slots as our equivalent of the box (Figure 11).

The empty-slot convention does have a pragmatic problem: Blocks that have default input values in their palette won't work in functional input slots unless the user explicitly erases the default input. For example, if you want to use the 

block to form the plural of a word, it's not good enough to replace "world" with an "s"; you must also delete the "hello." (Figure 12) We do also continue to support explicit formal parameters, but our teaching materials start with empty slots.

Figure 9. Computing the function input to `call`.

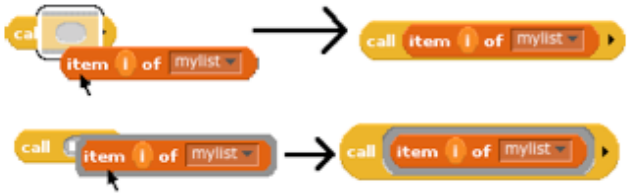

Figure 10. Using drop distance to control encapsulation.

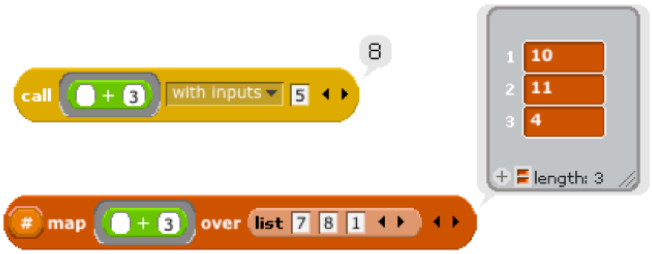

Figure 11. Empty box as implicit formal parameter.

III. From BYOB to Snap*!*

In 2011 we started planning for the next major release, BYOB 4. Partly influenced by the Scratch Team's plans for Scratch 2.0, we decided BYOB 4 should run in a browser. Meanwhile, however, we heard vocal complaints from a few teachers about the name BYOB, and we reluctantly agreed to change it. Unfortunately, we couldn't find an equally clever alternative, so we named the new version after the act of snapping blocks together to make a script. (The new name has a backronym: Software for the New Advanced Placement course.) Snap*!* 4.0 was released in 2013.

A. *Taming the Grey Ring*

One of our biggest design concerns was to undo the distance-dependent ringing of expressions. We had two somewhat contradictory goals: to handle every possible case, and to allow users to use higher order functions with essentially no knowledge of encapsulated procedures, and with no arcane maneuvers.

The ultimate solution was to try to make the common use cases really easy, even at the cost of (1) an evaluation model that's hard to explain in its entirety and correctly, and (2) requiring more user sophistication for the less common cases.

As a simple example of the first point, we decided that blocks or scripts dropped into a procedure-type input slot are *always* ringed, *unless the block is a variable reference.* Variable blocks are not ringed. Although it's possible to imagine wanting to delay evaluation of a variable (e.g., because its value will be changed later), the most common case

Figure 12. The user must explicitly empty the left box in `join`.

Figure 13. Using a reified `item` block as the function input to `map`.

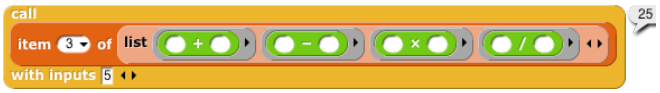

Figure 14. A non-reified `item` block computes the function input.

of putting a variable into a procedure-type input slot is that the user is writing a higher order function, and the variable is a formal parameter of that function, representing a functional input. In that case, the procedure to be called is the one that's the value of the variable, not the variable block itself.

The item of block is a difficult special case. Most of the time, it should be ringed, because the user's intent is to invoke the `item` block itself as the anonymous function (Figure 13). But if the list input is a *list of functions,* then it makes sense to evaluate the `item` block right away to determine which function to call (Figure 14). This is one of the use cases we decided would be uncommon enough that we could require extra sophistication on the part of the user, who must drop the `item` block into the `call` block (ringed, automatically) and then right-click it and select "unringify" from its context menu.

This last example also shows that there are still situations in which the user must explicitly ask to encapsulate expressions as functions: The `list` block takes inputs of any type, and so it would ordinarily take the arithmetic operator blocks as expressions (meaning, as in Scratch, 0+0, 0–0, etc.) to evaluate. To construct a list of functions, the user must either choose "ringify" from the context menu or drag a ring from the palette; they're now at the top of the Operators palette.

What if the user ringifies a block and *then* drags it into a procedure-type input slot? Does it get ringified again? We decided that in such a situation the user is almost certainly

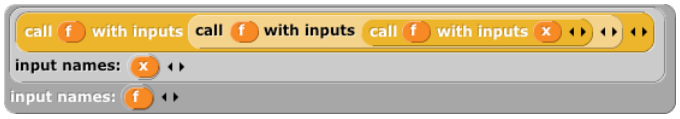

Figure 15. The Church numeral three.

Figure 16. Two nested `map` calls to process a matrix

confused about how Snap*!* handles functions as arguments, and does not actually want a double ring. So in this situation the two rings are "absorbed" into one. There *are* uses for doubly-ringed expressions, but they're rather advanced; an example is the definition of Church numerals in λ calculus, in which the number 3 is represented as

$$\lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,x))$$

In Snap*!* this would be rendered as in Figure 15. More common, but unproblematic, is the case of rings that are nested but with blocks separating them, as in Figure 16. (Note in passing that both kinds of nested rings really call for explicit formal parameters to make the code readable, even though the matrix example would do what the user intends even with empty-slot implicit inputs.)

Church numerals aside, though, the main result of this redesign is that a naïve user can use higher order functions even without understanding what the rings mean! The ring for a functional input is now built into the input slot itself, as in Figure 17. The user drags an expression into the ring, and is unsurprised that the ring is still visible. Using the empty slot notation for implicit parameters, it's easy to construct Figure 18 and just read it as "map plus-three over numbers."

IV. Still to Come

We haven't yet come up with a satisfactory (to both of us) solution to the problem of blocks with default input values.

More excitingly, we are planning in the next version of Snap*!* to have self-reflection for programs, so that we can implement the equivalent of Lisp macros. This will complete the project of smuggling all the major ideas of Scheme into Scratch.

Figure 17. The map block provides a grey ring where a function is expected.

Figure 18. "Map box plus three over numbers."