

## Hiding and Showing in Snap! Pedagogy

version 1.3, 4/19/23

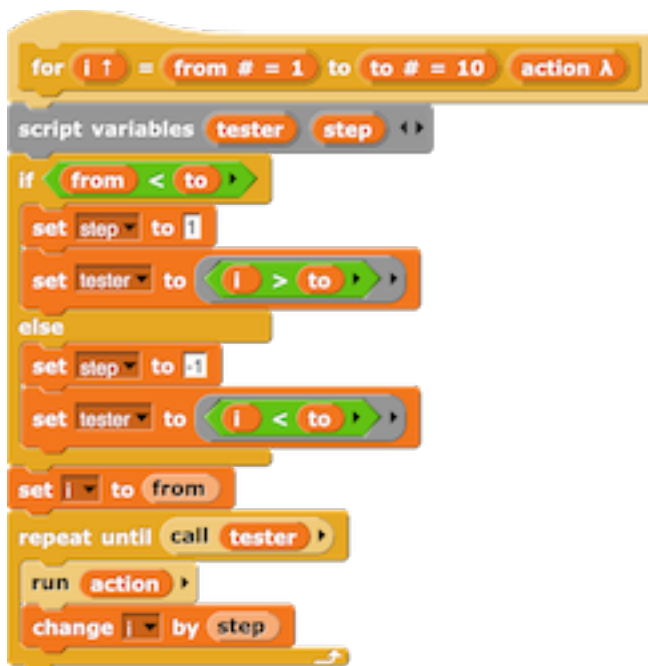
*Abstract.* I think that in recent years there has been a change in how we think about the pedagogic role of Snap!, perhaps not intentionally. It has to do with what is hidden and what is shown, broadly speaking. I don't know whether the shift is good or bad; this is not a Policy Proposal, just an invitation to discussion.

*Background.* I set out to write a more straightforward document, a request for two backward-compatible changes to primitives to correct what I see as bad design decisions, mostly by me, long ago. Here they are:



Almost all the time, if the starting value is greater than the ending value, you don't want to run the code in the loop. The canonical example is FOR I = 1 TO (LENGTH (some list)). If the current FOR would run backward (step size -1), the list is empty, and you don't want to run the body of the FOR at all. Similarly, you want NUMBERS to report an empty list.

I knew all that back when I wrote the version of FOR in the BYOB Tools sprite. So why didn't I do it as UP TO? Because at that time we wanted the *code inside* the tool blocks to be pedagogic. That's why the HOFs had to be implemented recursively, even if an iteration over a Javascript array would be quicker, for example. And that's why I wrote FOR like this:



The point of downward-stepping FOR was merely to show off the use of a ringed predicate stored in a variable, in a reasonably convincing context.

Since we no longer have that implementation of FOR, why don't we do it right (i.e., stepping upward)? Well, because of backward compatibility, and because Jens has applications for which he finds the current behavior useful. So, let's give users the choice. (We can fight later about which option should be the default. ☺)

And then I thought, "why am I asking for this kludgy menu instead of asking for a second UP TO block for FOR and another for NUMBERS?" And that brings us to the beginning of the discussion I want to have.

*Hiding and Showing.* Many of the design decisions we've made are about whether or not to show the user some detail. Here are some examples:

1. Radically hiding primitives so that students see only the ones you want them to use in this particular exercise (Parsons problems), to focus students' attention on just a few blocks.

1½. For the first several years of BYOB, though, we adamantly refused to implement hiding primitives, because on principle we didn't want to hide *anything* from the user. What changed our mind was Paul Goldenberg wanting the feature to teach *math* (not computer science) to *early childhood* kids (not Operational ( $\geq 8$  years old) kids). (And since then other teachers with similar situations.) Details that are intellectually rich for a CS student can be distracting clutter to a math student.

2. Hiding (behind Relabel) variants of primitives that are great exercises to implement in Snap! ( $\leq$ ,  $\neq$ ,  $\geq$ , min, max, etc.), to justify the exercises by pretending those functions aren't already provided. In particular, we do this for functions written in BJC exercises.

2½. But we're both kinda embarrassed about this neither-fish-nor-fowl compromise.

3. Unevaluated input types, in the first instance, to implement reporter IF in a way that made both us and the users happy. Listed here because the point is to keep the function-ness of an input hidden:



If the IF reporter is written in Snap!, then the THEN and ELSE inputs are really functions, not values, but they can't look like functions (in rings) because users might not know about those yet (and because Those Other Languages hide the delayed evaluation in syntax).

3½. We remain proud of this one! It lets our users invent their own control structures, and, importantly, *we* aren't hiding anything from *our* users; rather, our users are hiding the special formness of their procedure from *their* users. That sounds like a quibble, but it's not; it's not disempowering because next time it might be *that* kid writing a special form.


4. We limit the length of the palettes by relegating some things that maybe could be primitives into libraries, which makes them less discoverable, but exposes the fact that you could write them yourself in Snap! itself. Read that sentence again: hidden in one way, but exposed in another.

4<sup>1/2</sup>. This used to be an absolute rule; anything that could be written in Snap! *would* be written in Snap!, and put in a library. Examples include, most notably, the higher order functions on lists—arguably, the whole point of BYOB was to let kids write HOFs themselves—but also FOR, ASK, TELL, CATCH, THROW, and many more. We boasted that we added just eight primitive blocks to Scratch. But over time several of the library blocks have trickled into primitive status, partly for speed of execution and partly to have them ready to hand when starting Snap!.

4<sup>3/4</sup>. This policy shift is connected, in subtle ways, with the fact that the hyperblock feature has made HOFs less crucial for using lists, so we worry less about how we present them to users. But arguably that should have made us more willing to let the HOFs themselves run at Snap! speed rather than at JavaScript speed, rather than more willing to hide their implementation.

4<sup>7/8</sup>. We have talked about, but not implemented, *hybrid blocks*, which run fast like primitives, but have an editable user-visible definition like custom blocks.

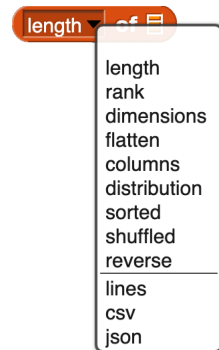
5. Another way we limit the length of palettes is by cramming several operations into one primitive block. For example, one of the obvious missing features in Scratch (as a deliberate design decision on their part) is the ability to send a message to a specific sprite. We added that,

but instead of a  block, we put this in the same BROADCAST block:




The variadic input(s) let us present the bare-bones, just-like-Scratch version, but with the subtle arrowhead to suggest that there are variant versions to be discovered. (But maybe the arrowhead means that you can broadcast more than one message at once? That's what it usually means: a variadic input.) And this subtlety just reduces the palette by *one* entry. Is it worth it?

5<sup>1/2</sup>. The *reductio ad absurdum* of this technique is the LENGTH block:

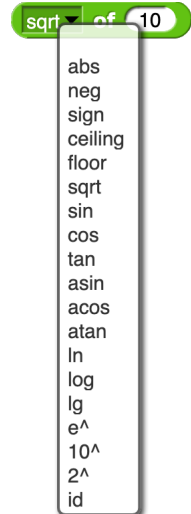


A dozen different functions, quite different from each other, and only the first three arguably related to LENGTH. It saves a ton of palette space! But really LENGTH is the only discoverable one; most of the others won't be obvious even if you see their names. What's the RANK of a list? Does FLATTEN flatten one level, or all the way down? And what on earth is

DISTRIBUTION? When we learned from APL to think of a list of lists as a multi-dimensional array (such as a matrix), we thought, "what's the multi-dimensional equivalent of LENGTH"? And APL's answer is DIMENSIONS ( $\rho$ ); RANK is just an abbreviation for

 ( $\rho\rho$ ). So those first three arguably go together. But the others are connected with LENGTH only in that they have arity one and have lists as inputs. This structure should be rethought, we're agreed.

5.51. What justifies this bizarre hiding of primitives behind a menu of names is the transcendental function block we inherited from Scratch:

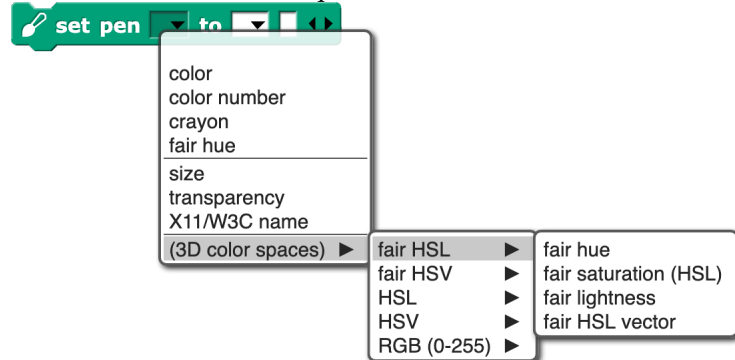


But this is already hiding more primitives than in Scratch. We added eight alternatives to the original ten, almost doubling the length of the menu. And the original ten functions all have numbers (including "Infinity") as their domain and range, computing (except for SQRT, which is algebraic) transcendental functions. By the way,



is disconcerting. It might be better to use " $\infty$ ," even though it may be unfamiliar to some young users, because we can more plausibly treat that glyph as a digit.

5<sup>3</sup>/<sub>4</sub>. But I still think the portmanteau



makes sense. It has a lot of options, but (except for SIZE) they're all about controlling the pen color, and there are many ways to think about pen color because there just are; it's not our doing. And all the variants are about SET PEN, which is this block's name.

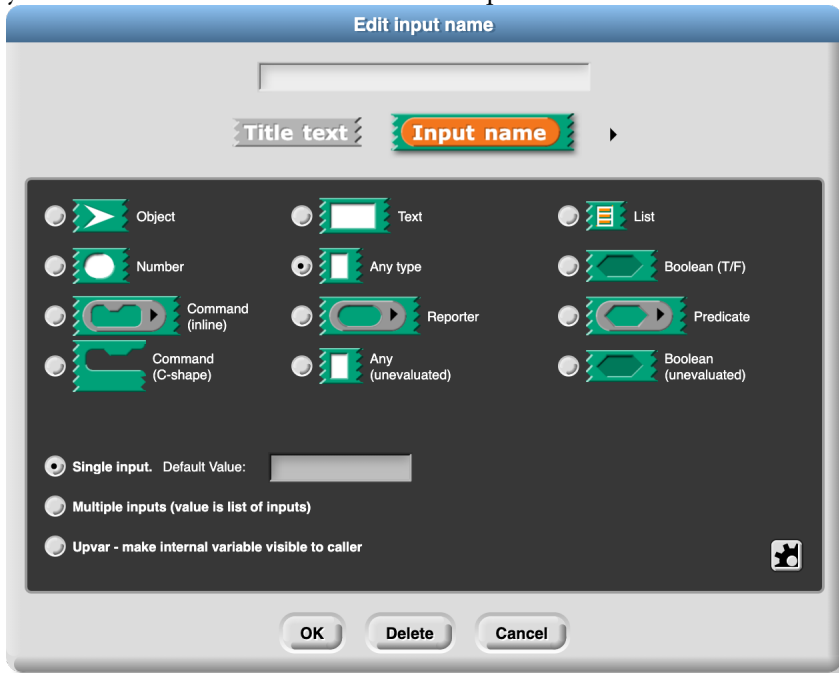
6. There are still things that work in primitives but don't work in user custom blocks. One example is grouping title text for an input with the input itself, in a variadic infix block:



As for any variadic input, clicking the right arrowhead adds an input slot. But it also adds the plus sign, as title text before the new slot. Saying this another way, the right arrowhead adds a *group* of things (two things, in this case) to the block. Users want to be able to do that in their blocks, too, but so far they can't. This is a kind of hiding; we want to keep the custom block input type dialog reasonably simple. In fact, the default UI to add an input slot is *very* simple:



That works fine for beginners. If you decide you want to specify a type for your input slot, then you have to advance to the more complicated version:



By the way, the arrangement of types in that display isn't random; each row and each column represent a category, although I may be the only Snap! user who's consciously aware of that, on the assumption that nobody actually reads the manual. Still, the arrangement simplifies finding the obscure types and connects each Unevaluated type with what it means (just above) and with what it looks like (two rows up, except for Commands). So, we try to reduce the cognitive load somewhat, while still giving users some flexibility. But, as an exercise, try to design the ability to specify a variadic *group* of inputs, both how the users indicate a group in the "Create input name" dialog and how the user of a custom block knows that the right arrowhead will add a group, not just a single input slot. So far we haven't tackled that hard problem. We're very proud of the simple input name dialog, along with a Settings option to start with the long form each time, for more advanced users, but still *kinda* simple.

7. On the other hand, we are about to unveil a radical un hiding: showing users the innards of expressions and procedure bodies via *macros* and *metaprogramming*. The central feature making this possible is the ability to convert back and forth between executable code (the blocks and scripts that we've always had) and *syntax trees*: lists of lists of the individual blocks and constant values (such as text and numbers), the building blocks of expressions and scripts. That conversion overcomes the only weakness of block languages, namely, that programs aren't data, which makes them harder to manipulate programmatically.

7<sup>1/2</sup>. Pedagogically, this is an extension of the self-reflection by means of non-hidden continuations, which call attention to the sequence of events in executing a script, and which we inherited from Scheme. Continuations are conceptually simple for the implementor, since they already exist in any interpreter for any language, and it's just a matter of making them visible to users. But they're not conceptually simple for users! The fact that they can be called repeatedly, and from outside of the script in which they were created, feels like magic. And it is; it's the magic of "everything first class." But the point here is that we didn't add this feature in response to a specific pedagogic need. Rather, explicit continuations help advanced programmers write advanced programs, and a few library blocks. And they're a way to plant the flag of Scheme in Snap!, which was a big part of our motivation.

8. Dating back to BYOB, we have two internal representations for lists: as Javascript arrays and as linked pairs as in Lisp. The reason is that certain functions of lists that are implemented recursively have  $\theta(n^2)$  asymptotic behavior for arrays but  $\theta(n)$  behavior for linked lists. It turns out that  $n$  has to be *very* large for the asymptotic behavior to outweigh the constant-factor advantage of using the primitive features of the implementation language. Still, here we are with two internal representations of lists. This is an opportunity to teach the first lesson of a data structures class, but we deliberately pass it up, taking pains not to show the user which representation we use for any particular list. That choice reflects the desire to have lists "just work" for users, without having to know their implementation. (The implementation breaks through, however, when the user mutates the list and runs into the fact that linked lists can share data.) By hiding the implementation, we in effect stake out the position that we know what's good for you better than you do. I take the full credit and/or blame for that choice. Its pedagogic intent is to lower the barrier to entry for lists, partly because they were considered difficult and esoteric by Scratch users, in the early days.

9. Dating even earlier, we've inherited from Scratch the grammatical specialness of numeric input slots. They're round instead of rectangular, and the characters you can type into them are restricted to digits, a minus sign in front, and the letter "e." (You can actually include more than one "e," but if you do arithmetic on such a supposed number, you get NaN as the result.) Every so often someone suggests that we accept "0x" at the front and digits a-f, but the real question is why we have syntactic restrictions in these input slots at all. Of our dozen-odd data types, why are only numeric slots restricted at the level of typing inputs into input slots? When you load the library that implements the complete Scheme numeric tower, including bignums, exact rationals, and complex numbers, you find yourself wanting to type "2/3" or "4+3i" into a numeric input slot and not being able to. The syntactic restriction doesn't work anyway, because you can put any function call into any slot, and we don't know what type of value your function will return

until it does so, and so there's also a semantic restriction when arithmetic expressions are evaluated, and so the syntactic restriction is redundant. Mostly the story we tell about input slots is that their visual type indicators (round slot for numbers, gray ring for procedures, stacks of elements for lists, etc.) are advisory, not enforced.

There is a somewhat better answer to "why?" than just "because Scratch." Numbers and strings are the only data types that are entered directly through the keyboard. So you can't type, say, a number into a list slot, because you can't type *anything* into a list slot. You have to drag a reporter into the slot; to use a constant list, you use a LIST reporter. Typing a non-numeric text into a numeric input slot is the only syntactic restriction that wouldn't exist if it weren't explicit. From this point of view, Snap! is already more syntactically rigid than it seems at first glance. But, for example, we don't take advantage of the hexagonal shape of Boolean input slots to limit those slots to accepting hexagonal blocks, as Scratch does. Scratch doesn't have custom reporter blocks, so it's only a handful of primitive predicate functions that can fit into a Boolean slot. But we can have user functions that return, e.g., a list, if one is found that satisfies some condition, or False, if not. So we have to accept round reporter blocks in hexagonal slots.

What's hidden or shown here is syntactic restriction, if you tend toward wanting no restrictions; or semantic types shown through syntax, if you tend toward wanting Java-style type declarations. Dan Garcia has been a strong advocate for (optional) type checking. On the other hand, hyperblocks allow list inputs in pretty much any scalar input slot, so one data type would be "lists in which every element satisfies the same condition"—lists of lists of ... of numbers, rather than just plain numbers. (But a just plain number matches *any* shaped list.)

*Some tentative conclusions.*

When we started working on BYOB3, our straightforward goal was to give kids access to pretty traditional computer science: algorithms, data structures, programming paradigms, recursion (not just as looping), and so on. That's still the goal of BJC, which developed in parallel with BYOB. But another educational strategy, exemplified by media computation, is to teach application-driven topics. Even in BJC, there was originally a page called MAP, a page called KEEP, and a page called COMBINE; now, instead, there's a Contacts app like the one on a phone (albeit simpler) and the higher order functions are introduced as needed to maintain the contact list. The SAP-housed curriculum is even more application-driven. The changes around hiding vs. showing are largely situated in the larger change from big-idea-driven to application-driven. But it's crucial to remember that that change isn't just in CS *education*. Since the sudden rise in Data Science as an intellectual enterprise, research computer science itself is more and more application-driven, guided by collaboration with non-CS faculty. Technically, the computer science is largely embedded in statistics. So what's happening in Snap! curriculum isn't a struggle between CS and applications; it mirrors changes in CS itself.

To a first approximation, hiding is good for beginners; showing is good for more advanced users. In one case we've explicitly designed for both cases, namely the input dialog with its short and long forms. (See #6 above.) In most other cases, we've made one choice, often based not on principled reasons but rather on courage (showing) or timidity (hiding) about how much users can understand. We should consider whether dual designs could help in other cases as well.

**Making the palette smaller isn't a good reason** to increase the complexity of blocks. We know a better way: letting users show or hide a palette subcategory explicitly in the UI, with arrowheads in front of subcategory names. (In boldface because I feel quite confident about this one, although Jens isn't convinced.)

More generally, we should agree that things in dropdown menus don't count as discoverable.

The long form input dialog can be made the default with a Settings checkbox. If we extend dual design to other areas of the implementation, we run the risk of greatly expanding the Settings menu. Maybe menus, too, need subcategories that users can show and hide.

But mainly, we need a discussion of a general policy for hiding and showing, rather than deal with each specific case as it comes up.