An AST-based Interface for Composing and Editing JavaScript on the Phone

Yana Malysheva
Google, Inc.
San Francisco, CA 94105
Email: yanamal@google.com

Abstract—**Grasshopper is an Android application which teaches people JavaScript through a series of coding puzzles. As part of this application, we developed an AST-based JavaScript editor for the phone. The visual representation of the code is built using the AST as the source of truth, and user actions modify the AST rather than a code string, much like a block code editor. At the same time, we style this visual representation to look as similar as possible to syntax-highlighted, formatted JavaScript code in a text based editor. This paper outlines the system, the reasoning behind this design, and some early observations from users interacting with it.**

I. Introduction

Grasshopper is an Android phone application which aims to introduce adults with no coding experience to JavaScript and computational thinking through short, self-contained programming puzzles. Developing a smartphone app allows us to reach a larger audience than a computer application would - primarily because smartphone ownership is overtaking computer ownership both in the US and in emergent economies [1] [2] [3], but also because people have access to their phone in more situations than their computer (on their commute, waiting for an appointment, etc.). However, developing a code-editing interface for the phone, especially one that is accessible to beginners, presents some interesting challenges.

In order to effectively teach JavaScript to beginners, we wanted to make our visual representation of the code look very similar to JavaScript in a "normal" text editor. We also wanted our editor to be both easy to use, in that composing code is as simple and fast as possible, and intuitive to learn, in that the interface itself would not have a difficult learning curve.

At the same time, we wanted to retain some of the benefits of block code interfaces: we wanted it to be very hard for users to make syntax errors, and easy to gain an implicit understanding of code structure through using the editor to compose and manipulate code.

The resulting system looks similar to a traditional text editor but behaves a lot like a block code editor. We believe that these characteristics are interesting even outside of the phone app context.

II. Related Work

There has been a lot of recent work around hybrid interfaces that close the gap between text-based and block-based code editing by either representing existing text languages as blocks (e.g. Droplet [4] [5], DrawBridge [6], Code Kingdoms [7]), or representing block-like languages in a more textual way (e.g. GP [8]). This is usually achieved by using the Abstract Syntax Tree (AST) of a program as a single source of truth, but allowing the visual representation and interaction with the program to vary along a text-block spectrum: Droplet and DrawBridge allow the user to switch between the syntax-highlighted text view and a decorated version of the same code that looks like block code. GP and Code Kingdoms additionally allow for several intermediate states along the block-text spectrum.

Greenfoot's Stride editor [9] takes a different approach to the idea of a hybrid between text-based and AST-based editing: language constructs and program structure are represented as frames, and input is restricted to make it very hard to create syntactically invalid code; but the main form of input is typing in text (and selecting from auto-complete-style options, when appropriate), and the resulting visual representation looks very similar to syntax-highlighted text.

Wendel and Medlock-Walton [10] have pointed several potential benefits to using ASTs as the source of truth, in addition to syntax-error-reduced editing and easy transitions between representations: for example, real-time collaboration and source control could both be much simpler and more powerful if they used the AST rather than the textual representation as the underlying model.

There have also been some recent developments around mobile coding environments (TouchDevelop [11], Hacked [12], Swift Playgrounds [13], Hopscotch [14]), though notably many of the existing publicly available mobile apps are designed primarily for use on tablets rather than phones, in order to take advantage of the increased screen space. A few of the available apps (Hacked, Swift Playgrounds) appear to use a token-based code editor with context-sensitive suggestions, rather than using a program AST as the source of truth. Others rely on a custom block-like visual language.

Grasshopper's code editor aims to combine advances in these two areas by having a touch-based phone interface that executes code in a preexisting language (JavaScript), uses the AST of the program as the source of truth, but presents the JavaScript code in a recognizable text form. It also aims to optimize for a code composition process that is both fast and intuitive for beginners who may not have encountered any kind of programming language before.

Interestingly, the phone interface inverts the traditional priorities of a hybrid AST-based code editor somewhat: A lot of the recent research has been around ways to allow for keyboard input in a block-like environment in order to increase code composition speed. But on a phone, trying to use the software keyboard can actually quite a bit slower than a touch-based block-like editing interface.

At the same time, because we are targeting adults, our audience is likely to have at least some experience with editing and composing text on a computer or mobile device, but not necessarily with using block code editors. Seeing code represented in a text-like format is actually more likely to be familiar to them than a block format. So, by using a text-like display, we hope to present an easier learning curve to our students - both when they are learning how to compose code, and when they are transferring their JavaScript skills outside of the Grasshopper environment.

A text-like representation of the code may also make the interface feel more authentic to learners, especially adults or older children. Research suggests that people perceive text-based coding as more authentic and powerful [15] [16], and our own user tests suggest that adults are often reluctant to experiment with a system unless it feels like they are immediately getting something "real" out of it.

Finally, as noted in [8], block-like representations of code tend to use up more space than text-like representations, so most visual representations of code structure would exacerbate the screen real estate problem that already makes coding on the phone quite tricky.

III. System Overview

A. General Structure

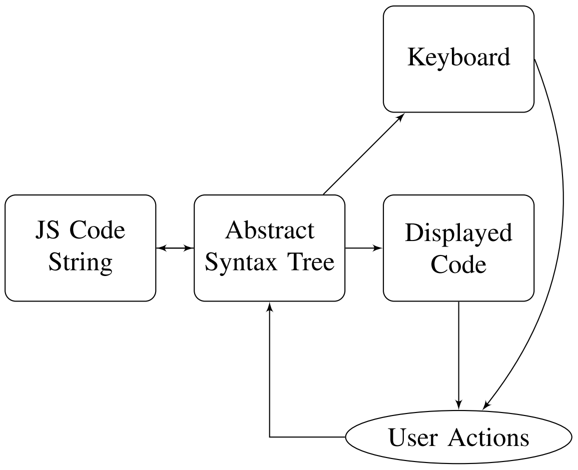


Figure 1. Overview of code editing flow

The student code is stored in our database as a string of JavaScript code, just as a regular text editor would create.

When the application needs to display a given piece of code, it first parses it into an Abstract Syntax Tree using a standard parser (Esprima [17]). This AST is then augmented with the addition of special "Placeholder" nodes in places where additional code can be inserted.

The app walks through the modified AST and generates a list of tokens to display a representation of the code. These tokens retain a reference to the AST node they correspond to. Tapping on a token selects the subtree rooted under the associated AST node. Standardized formatting such as newlines and indentation is also encoded as tokens. The tokens are displayed as just text (without any visual block like distinction), so the final result looks very similar to syntax-highlighted code in a text editor.

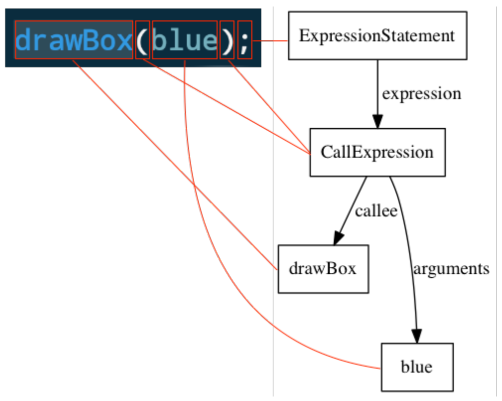


Figure 2. An example of a mapping between displayed tokens and the underlying AST

At the same time, the AST of the code is also used to generate a list of required keys and populate a special code keyboard. Each key specifies an AST node or subtree that can be inserted into the code. The code keyboard is populated such that it is possible to use it to recreate the specified AST. In addition to the AST of the current on-screen code, the system can also take in additional configuration if more keys will be needed, e.g. the AST of the canonical solution to the current puzzle; or a manual specification of the desired keys. For example, the experimental "sandbox" mode takes in a long configuration of a carefully-selected subset of JavaScript language features, which allows for the composition of fairly complex code.

The user edits the code by interacting with the keyboard and displayed code. The user's changes are then applied to the AST, and the keyboard and code display are updated as necessary to reflect the changes. So, for example, if the user has declared a new variable, this variable will now be available in the code keyboard to use in other places in the code.

To save the user's progress, we encode the AST back into a JS code string, and store the string in the database. This code string is also used for executing the code.

B. Editing Interface

The user manipulates the code using two types of actions:

1) Tapping on a token to select the associated AST node, and the entire subtree under this node.
2) Tapping on a key in the code keyboard to replace the currently-selected subtree with a different subtree. (In

some cases, the code that was replaced is reused in some way in the new code - see optimizations below).

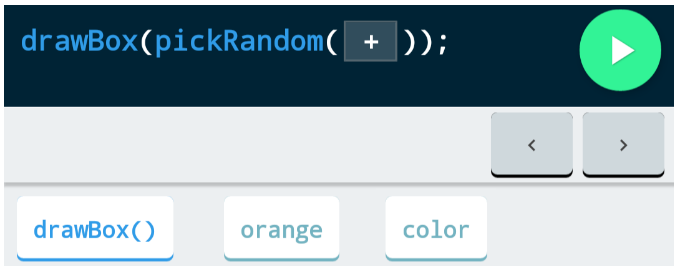

```
drawBox(pickRandom( + ));
```

Figure 3. Part of the code editing interface, showing the code view and the keyboard.

Adding or appending code, then, is simply a case of selecting an "empty" placeholder node in the AST and then replacing it with the subtree specified by a key. (See Figure 3) Similarly, deleting code (by tapping the delete key) is a case of replacing the selected subtree with a blank subtree.

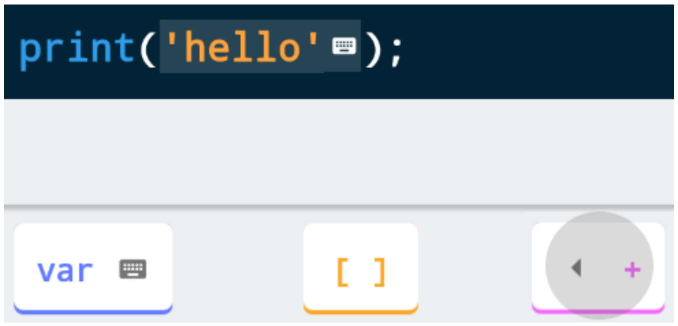
C. Optimizations for ease-of-use

Building on top of the general structure described above, there are several categories of special-casing that optimize for fast and intuitive code editing.

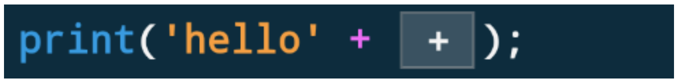
1) Selected subtree reuse on code insertion: Generally, when the user taps a key, the current selection is replaced entirely with a different AST node or subtree, as specified by the key. This can be thought of as composing code "top-down": To compose some code, the user first specifies the topmost node, and then recursively specifies or modifies its children as needed.

However, in some cases, we add custom logic which reuses parts of the replaced node or subtree. Broadly, this reuse falls into two different types:

- Reusing the original selection as one of the children of the newly created node. This is used, for example, for binary operators: in a top-down composition mode, to compose an expression like "x+y", the user would need to tap the "+" key to insert the operator, and then specify each operand after the operator is already in place. However, in a text-like environment, it is much more natural to compose expressions like this left to right, especially for beginner programmers. So, in our system, to compose "x+y" the user would first insert "x", then, with "x" selected, press the "+" key. At this point, the "+" node would take the place of the "x" node in the AST, and the "x" node would be re-rooted as the left operand of "+". The user can then replace the (blank) right operand with "y".
- Retaining all or most of the children of the original subtree, and only replacing the root node. The main use of this is to allow for replacing a function name while keeping the same parameters. One could also imagine using this to change the type of a loop while retaining its body, or even to change an if statement to a while loop, and vice versa. Of course, this type of replacement only makes sense if the two node types are compatible, i.e. they have the same set of expected children types.


```
print('hello');
```

(a) immediately before tapping "+"


```
print('hello' + + );
```

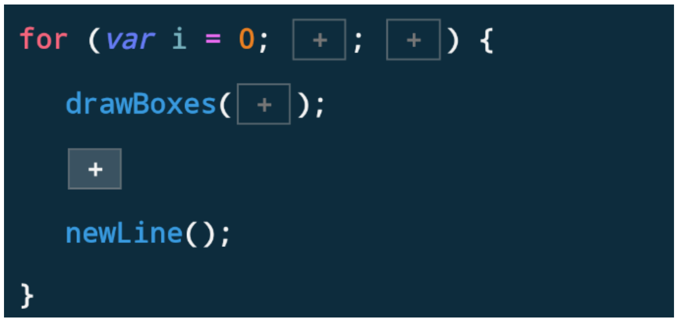
(b) Immediately after tapping "+"

Figure 4. Part of the process of composing a binary operation

2) Next Selection: Every time code is added or deleted, the node selection changes (since the previously selected node no longer exists in the same place). The logic for what to select after an insertion or deletion is heavily dependent on the type of node that was just inserted/deleted. This logic tries to minimize the number of times the user has to manually change the selection while performing common sequences of code edits. In other words, we try to optimize for editing code by tapping a sequence of keys on the code keyboard, without having to adjust *where* you are inserting/deleting code. For example:

- After the user inserts a function call, the selection moves to a blank placeholder in place of the function's first parameter (unless the function is known to take no parameters, in which case it moves to a blank placeholder that would allow the user to start a new statement.)
- After the user deletes a statement, the selection typically moves to the previous statement in the block. If there is no previous statement, because the deleted statement was the first statement in the block, the selection moves one level up to the statement that defined the block, e.g. an if statement or a function definition.

3) Minimizing the number of visible placeholders: Normally, users insert new code by selecting a blank "placeholder" node (represented on screen as [+]) and replacing it with the desired code. However, showing a placeholder symbol in every single place where insertion is possible is both overwhelming and wasteful of the very limited space a phone screen allows. Thus, we remove the placeholders from view in many common places, unless the user is already in the process of editing in that particular location. Instead, we add invisible touch targets in these locations. Then, if the user wants to add something to a location where no placeholder is shown, they tap into that location as if they are moving the cursor there, and we show



```
for (var i = 0; [+]; [+]) {
   drawBoxes([+]);
   [+]
   newLine();
}
```

Figure 5. Code with several placeholders: the highlighted placeholder is "transient", in that it will be removed if the selection moves away from it. The other placeholders indicate code that still needs to be filled in.

a (selected) placeholder in that location, which they can then replace by tapping a key on the code keyboard. If they move the selection away without adding anything, we once again remove that shown placeholder.

We primarily do this for placeholders that enable inserting a new node in an ordered list: a sequence of statements in a code block, a sequence of items in an array, a sequence of key/value pairs in an object definition, etc.

We never do this for placeholders that represent "incomplete" code, such as a missing parameter in a function call. So, the user can clearly see where they still need to add code.

The overall effect is that the placeholders act a lot like a cursor in a text editor. This seems quite intuitive for new users: we never explicitly explain this mechanic, but most users have no problems using it to navigate code.

IV. Results

We have conducted a number of in-person qualitative user studies with Grasshopper in both group and individual contexts, observing between 20-30 users in total. We have also released the app for beta testing, and analyzed user logs from beta users. At the time of writing, just over 2500 beta users have downloaded the app, and just under 1000 have logged in and started interacting with it.

In general, the results are very promising: many users with no coding experience are able to use the interface to compose code, even though we do not explicitly explain any of the aspects of the interface - we just present the user with coding puzzles of increasing difficulty.

Below are some specific encouraging observations, as well as some interesting limitations and side effects we had not anticipated.

A. Observed Advantages

This type of editing interface retains some key advantages of both block code editing and text editing. Our interface in particular has also benefited from being developed in tandem with the content of the app, so that they support each other: the interface is optimized toward making the puzzles easier to solve, and the puzzles are optimized toward scaffolding the interface functionality.

1) Manageable space of possibilities: As with all block- and AST-based interfaces, it is very hard (though not entirely impossible) to make syntax errors in our system. Therefore, almost any action a student takes tends to produce at least some interpretable result, rather than a syntax error message, which for beginners usually only contains one bit of information: "No, that was not correct".

In our specific implementation, we are further limiting the space of possibilities (and therefore the chance of creating nonsensical programs) by limiting the keyboard, by default, to the keys required to solve a given puzzle (although we can and do override this in some cases with additional keys that are likely to create interpretable but incorrect output).

These features together make learning by experimentation feasible, as they allow for a fruitful feedback loop of editing code and seeing what it does, then editing it again to try and move toward the current goal.

2) Interface Ease-of-use: We have found that most users are able to navigate the interface and compose code without any explicit tutorial.

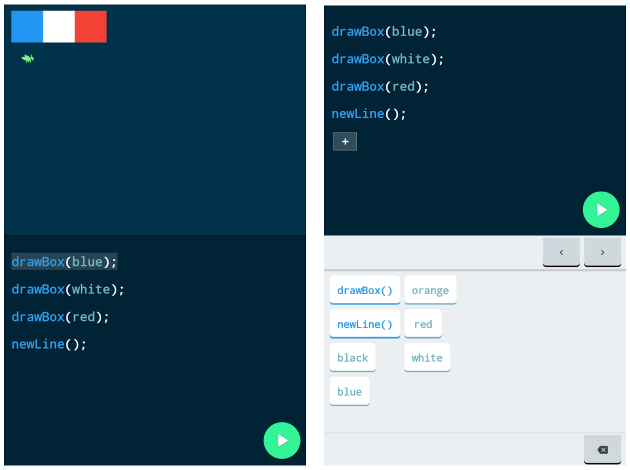
The first puzzle in our app asks the user to draw the lower half of a French flag by repeating the code that drew the top half. The user is presented with the coding interface, and the output of the starter code (i.e. the top half of the flag), but no explicit guide to how they should navigate the code, or which buttons should be pressed. All of our in-person user testers were able to figure this puzzle out with minimal encouragement (e.g. "try pressing one of the keys!", "Can you run your code and see what it does so far?"). In a recent sample of beta users (who have had no in-person help or interaction), 91% (640 out of 700) of users who made any code changes were able to complete the puzzle. In the same sample, 88% (640 out of 721) of users who **either** made a code change or re-ran the starter code completed the puzzle. (21 people in this sample re-ran the starter code, but never edited it; 18 other people opened the puzzle, but never interacted with the code or the run button.)

We think that grounding our interface in the familiar conventions of text editing helps the editor feel intuitive and not overwhelming.

3) Gradual reinforcement of code structure and abstract programming concepts: There are several specific programming concepts that we've found uncharacteristically easy to introduce using our interface:

- Variables: Because naming variables is one of only two cases where the user has to type in some text, we have seen much less confusion than usual around variable names as magic keywords. The users seem to have a solid understanding that the variable names are arbitrary, and do not affect the program's execution. In some cases, we have observed users spontaneously creating variables for storing intermediate results, even when it was not required to solve the puzzle (or anticipated by our system).

  Two particular aspects of our interface seem to further solidify this understanding: the fact that newly created



```
drawBox(blue);
drawBox(white);
drawBox(red);
newLine();
```

```
drawBox(blue);
drawBox(white);
drawBox(red);
newLine();
```

(a) The first time the user sees the puzzle (b) After the user taps somewhere in the code area

Figure 6. The first puzzle

variables instantly show up in the code keyboard (thus reinforcing the idea that you've created something that didn't exist before), and the fact that changing a variable name in its declaration statement also updates the name across all uses of the variable.

- Nested function calls: One of the early puzzles asks the user to **replace** a particular parameter in a function call with a call to the pickRandom() function, which chooses a random color. The user thus constructs the concept of nested function calls without being given an explicit definition or explanation of how to do this and what it means. The concept of nesting function calls is traditionally fairly hard to explain, but translates very naturally to a simple sequence of actions in our editor, since the code editing interface is very much centered around replacement. The user is able to translate the instructions into actions, and is then immediately able to see the effect of nesting function calls.
  Most users are able to complete this task: in our most recent weekly logs, 217 users attempted this puzzle, and 95.4% of them completed it. This is comparable to other early puzzles.

Thus, by asking the user to edit code through manipulations to the underlying AST, we are creating a gradual hands-on understanding and intuition around the code structure, and the relationship between the code text and the AST, without explicitly introducing the AST as a concept.

4) Content Development: The fact that the user's view of the code strongly correlates to the underlying JavaScript is very helpful from a content development perspective: when content creators are developing new puzzles, we are able to abstract out the specifics of our app's interface, and compose the content directly in JavaScript.

Because the code keyboard is configured by reading the solution and starter code of the puzzle, the content creator also does not need to specify the keys that are needed to complete the puzzle, and therefore cannot make mistakes by accidentally omitting required keys.

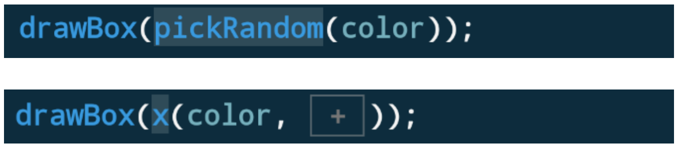
Thus, for the most part, content creators can be agnostic to the particulars of our application. In fact, it is entirely possible to develop a working puzzle without having tested it in the app. Though of course the quality of the content still benefits from some app-specific configuration, like specifying extra keys, or tailoring the user-facing feedback to refer to the app's UI elements.

B. Limitations and Edge Cases

1) JavaScript is a very flexible and loosely typed language: For example, on the AST level, JavaScript technically has very little distinction between function names and variable names: they are both considered Identifiers; in fact, in the course of a program, it is possible to reassign a variable to contain a function (and thus be callable just like a regular function) or vice versa.

But we try to make a distinction between function calls and variables in our editor, because for a beginner, it is very important to understand both concepts separately before trying to take advantage of how they can be mixed in JavaScript. So, we have to do a lot of complex reasoning about whether to treat a given identifier as a function or a variable, both in the code keyboard and in the syntax highlighting of the code.

Even with that, it is still occasionally possible to do unexpected and unintended things in our app, like accidentally substituting a variable name in place of a function name, instead of the intended action of replacing the function call with the variable. This kind of edge case is hard to catch, and requires more special-casing, because as far as the AST is concerned, a variable and a function call are the same thing.



```
drawBox(pickRandom(color));
```

```
drawBox(x(color, + ));
```

Figure 7. Before and after tapping "x", with the function name "pickRandom" selected. The intended result in this case is drawBox(x), but instead of replacing the entire pickRandom function call, the action has replaced just the function name.

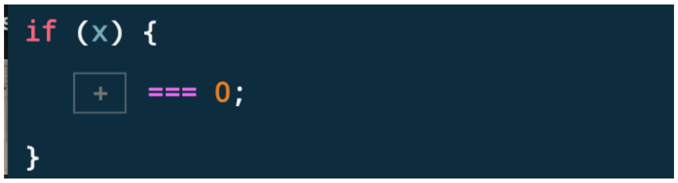
Because JavaScript does not have strict typing, errors like this (and others) are runtime errors, and therefore are hard to catch and expose to the user as at the moment the user makes the mistake.

Other language features, like objects and member expressions, are even harder to reason about, and allow for even more variations of nonsensical code that technically follows the JavaScript AST conventions.

2) It can be hard for a user to predict the result of a sequence of actions: As mentioned in the system overview, the selection state after an insertion or deletion is heavily dependent on the particular node types that were inserted and/or replaced, in order optimize for fast sequences of common operations. But because of this, sequences of two or more actions (without adjusting the selection in between) can also be counterintuitive and hard to predict.

For example, if a user inserts a binary operation, such as addition, but then immediately wants to delete it, it will take three "delete" key presses to get the code into a pre-insertion state: the selection moves to one of the operators after the insertion; after the first delete key press, it moves to the operator, then to the entire expression after the second delete, and only the third delete actually removes the expression.

Or, if you are composing an if statement, you might assume you can press the following keys in order without having to move the selection: "if", "x", "===", "0". However, this would produce the code shown in figure 8, because the app currently assumes you are done with the condition after you insert "x".



```
if (x) {
    + === 0;
}
```

Figure 8. The result of tapping the keys "if", "x", "===", "0" in that order, without adjusting the selection between key presses.

The opposite assumption - that the selection should generally stay inside a set of parentheses - breaks the flow of composing code made up of a sequence of function calls: in the current configuration, once the user is done specifying all the parameters, the selection moves to the next line of code, and is immediately ready to add a new function call. If the selection stayed inside the parentheses, the user would have to manually change the selection each time they wanted to start a new function call. In particular, this would make the very first puzzle (French Flag) harder, since the users would have to figure out several types of interface interactions at once.

3) Subtleties of the mapping between displayed code and the underlying AST: Because we do not directly introduce the concept of the AST, but do use it as the basis for navigating and editing code, it can be hard for beginner users to understand certain subtleties of what exactly they have selected. For example:

- Tapping the name of a function (in a function call expression) selects just the node for the callee name, while tapping on the parentheses selects the entire function call (See figure 2). Because of some of the special casing, the two types of selections act similarly, but not exactly the same.
  In particular, you can only append dot-notation expressions to function calls (e.g. foo().bar) by first tapping on the parentheses to select the entire function call. But most users of our app are accustomed to manipulating functions by tapping on the function name, and are unlikely to think of tapping on the parentheses as a separate, different types of selection.
  (see also function call replacement example in Figure 7)
- In a binary expression (e.g. "x + y") tapping on the operator only selects that operator. In order to select the entire expression, the user must tap on the whitespace around the operator. Again, actions on the operator produce very similar results to actions on the whole expression, so users are unlikely to notice this subtle difference until they encounter a case where they do act differently.
- Unary expressions are displayed without whitespace (e.g. "x++"), so it is actually impossible to select the entire unary expression by tapping somewhere in the code. It is, however, still possible to delete the expression by repeated use of the delete button.

More generally, the mapping between characters in the code and nodes in AST is not as strong or consistent as one might like or expect for this sort of application: because we are not using any graphical representation of program structure, we are relying entirely on this mapping to convey the AST structure. But this may get progressively harder with more complex language features, especially in a permissive language like JavaScript.

Most of the time, these unexpected behaviors do not stop the user from being able to do what they wanted to do, but they do add rather than reduce confusion about the underlying code structure.

4) Mixing top-down and left-to-right code composition paradigms: Our code editor does not currently display grouping parentheses in nested binary expressions. For example, the expressions "(x % 2) === 1" and "x % (2 === 1)" currently appear the same in our editor, as "x % 2 === 1". The standard order of operations dictates that this code should represent the first expression, but it is actually very easy to accidentally compose the second one: Typically, the user composes "x % 2", then selects the "2" in order to continue appending to the expression, and taps the "===" key. At that point, they've composed "x % (2 === [+])", but since the parentheses are invisible, they have no way of noticing that this is the wrong expression.

Of course, the omission of parentheses is a bug that can and should be fixed (as is the broader problem that our displayed code may not correspond to the executed code). However, even if we show the parentheses after the user has composed the wrong code, they will not necessarily be able to tell why they appeared there; let alone deduce how to compose the correct code (they would have to select the % operator or the entire modulo expression before tapping "===".)

If binary expressions were either composed top-down, or parsed as tokens left-to-right, this problem would not exist. But the subtle differences between the top-down order of evaluation and the left-to-right order of composition, compounded with the fact that the underlying AST structure is not immediately apparent, make this case very counterintuitive and confusing.

5) AST selection transitions can be counterintuitive: We have experimented with navigational buttons that would move the current selection around the AST: the forward/back button would move the selection to a sibling node in the tree, and the zoom in/out buttons would move up and down the tree.

Our hope was that by using these buttons to navigate the code, and observing the effects, the user would naturally gain an understanding of the underlying structure. However, the effects of the zoom in/out buttons on the selection were obfuscated enough that our users were not able to see the pattern in what they did, so they would stop using the navigational buttons altogether. In addition, moving to a sibling node with the forward/back buttons sometimes produces vertical movement, and sometimes horizontal. But our users generally expected to be able to move the selection left-to-right with one set of buttons, and up and down with another.

We eventually removed the zoom buttons, but kept the forward/back ones. Based on in-person user tests, we think that users generally avoid these buttons, and navigate by tapping on the code directly.

6) Rigid code styling: The code editor imposes its own style rules on the code, so neither the user nor the puzzle creators have direct control over things like whitespace and indentation. In most cases, this is a good thing, since it creates a consistent style, and we can optimize this style to be easy to read and manipulate on the phone. However, there are some cases where more flexibility or more nuanced styling logic would work better.

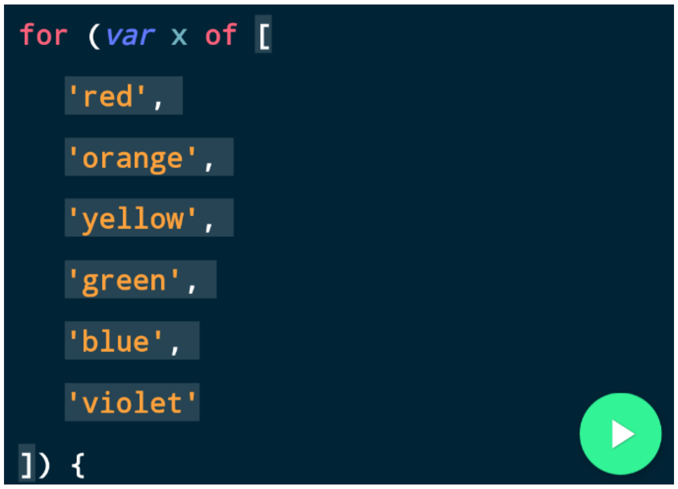
For example, arrays are always displayed with each element on a new line. This way, we can clearly demonstrate to the learner that each member of an array is a distinct entity, avoid messy line wrapping in cases of arrays of longs strings, and have large easily accessible touch targets between the lines for inserting new elements in the array.

But this gets very unwieldy once we start using 2D arrays in code: each element of each inner array is on a separate line, so the code takes up a lot of vertical space, and does not look very two dimensional - just a bunch of nested elements and square brackets. In this case, and perhaps some others, we would ideally display each inner array on one line of code, so that it's easy to see the two-dimensional structure of a nested array.

We're not yet sure what the solution to this particular problem is: how to make a good, consistent decision about whether to put line breaks between elements in an array.

7) It's not always possible to determine the set of needed keys from the puzzle definition: Normally, the system uses a combination of the current code and the canonical solution code to decide on a set of keys to populate the keyboard (as described in the system overview). For example, if a puzzle's solution code contains an if statement, then the system knows that an if key is needed in the keyboard. Although this heuristic is usually very good at automatically deciding on exactly the right set of needed keys, there are several cases where we've had to manually specify some additional necessary keys of a puzzle, or even rewrite the canonical solution in a certain way to ensure that the right keys show up.

One puzzle asks the user to print out a secret message contained in the variable "passphrase", to introduce the concept



```
for (var x of [
    'red',
    'orange',
    'yellow',
    'green',
    'blue',
    'violet'
]) {
```

Figure 9. Sample code with an array: each element is shown on a new line, to fit in the narrow amount of space available.

of variables as containers of data. However, the variable is declared outside of the student code (since its message is supposed to be secret), so the variable is never used in the starter code. Moreover, because the message is secret, and the output of the canonical solution is shown to the user before they start the puzzle, we have also altered the canonical solution to not use that variable - so the variable "passphrase" is not actually present in the card definition, and the algorithm has no way of knowing that this variable is important for the solution.

Variables have generally proven to be difficult to reason about: if we use the solution code to determine which variable keys to provide, then the algorithm will show keys for variables that aren't yet defined in the code at the start of the puzzle (the user is meant to define them as part of solving the puzzle, and the keys are meant to appear in response to the user defining the variables). On the other hand, if we ignore variables in the solution code, it's possible to omit some necessary variables that are defined elsewhere. Even if we tried to reason about variable declarations in the solution code, there are edge cases, like variables that already do exist in one scope, but are re-defined in the solution in another scope.

Another set of puzzles allows the user some freedom to use either a series of drawBox() function calls or a shortcut drawBoxes() function, which draws several boxes at a time. In order to have the keys for both versions available to the user, the canonical solutions actually use a very strange-looking mix of both functions.

V. Future Work

We are working on several major additions to our current implementation of the editor:

A. Key filtering based on current selection

Many block code editors stop the user from inserting the wrong types of blocks in places where they don't make sense.

They also typically give visual clues of what blocks can go where through their shape.

Although the current version of Grasshopper does prevent most incorrect code insertions by displaying a "That cannot be inserted there" message, it's still difficult for a user to know what can go where without trying an insertion and getting that message, which can be a frustrating process. In the future, we want to filter the available keys based on the currently-selected node, and the types of other nodes that can feasibly go there. For example, when the selection is on a function parameter, only keys that would produce a valid expression would be available; not "statement" type keys, such as the key to create an if statement.

B. Code reordering

We do not currently have an interface for reordering or copy-pasting code. So, for a puzzle where we ask the user to move a line of code to a different scope (inside an if statement), they actually have to compose the line of code from scratch inside the if statement, and then delete it from the original place.

In block-based interfaces with dragging as the main interaction, code reordering and code composition are almost the same action. But since our main code-composition action is insertion/replacement, we will need to make a separate interface, and possibly even a separate mode, for code reordering.

VI. Conclusion

This hybrid approach of visualizing code as text, but manipulating it like block code has thus far worked well for our application: beginners are consistently able to pick it up and start coding in JavaScript on the phone.

There are still some trade-offs to be considered and balanced: between the familiar text-editing metaphor and the subtleties of manipulating an implicit AST structure; and between optimizing for common editing use cases and having a generally more consistent, predictable interface.

So far, a fair amount of special-casing around making editing faster and more intuitive still seems necessary, especially with a highly permissive language like JavaScript. Our current system of special-casing grew somewhat organically in response to user testing, and right now we are trying to find a simpler set of rules that's good for both consistency and ease of use.

One of the hardest problems may be working around the relationship between the JavaScript text and AST: fundamentally, by displaying the code as syntax-highlighted text without any additional visual representation of structure, we are relying on the mapping between characters in the code and nodes in the AST to convey this relationship. Here, too, JavaScript's permissiveness exacerbates the problem: there are too many ways to do the same thing, and the same type of node often has too many different functionalities.

One possibility we are considering in order to constrain JavaScript's permissiveness is to treat the user's JavaScript code as TypeScript [18], at least in some contexts. This will allow us to reason more precisely about what does or does not make sense at code-composition time rather than runtime, and react more quickly to user errors.

Even with these open questions, we think that this approach is a promising option for editing code on the phone. One can even imagine it being used for coding outside of an educational context: existing developers also sometimes find themselves without immediate access to a computer, but want to jot down some code, or try out a quick algorithm idea.

References

[1] "Mobile technology fact sheet," *Pew Research Center*, 2017. [Online]. Available: http://www.pewinternet.org/fact-sheet/mobile/

[2] M. Anderson, *Technology Device Ownership, 2015*. Pew Research Center, 2015. [Online]. Available: http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/

[3] J. Poushter, "Smartphone ownership and internet usage continues to climb in emerging economies," *Pew Research Center*, vol. 22, 2016.

[4] D. A. Bau, "Droplet, a blocks-based editor for text code," 2015. [Online]. Available: http://ideas.pencilcode.net/home/htmlcss/droplet-paper.pdf

[5] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 445–448. [Online]. Available: http://doi.acm.org/10.1145/2771839.2771875

[6] A. Stead and A. F. Blackwell, "Learning syntax as notational expertise when using drawbridge," in *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014)*, 2014, pp. 41–52.

[7] "Code kingdoms," 2014. [Online]. Available: https://codekingdoms.com

[8] J. Monig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in gp," in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, Oct 2015, pp. 51–53.

[9] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing," *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, 2017.

[10] D. Wendel and P. Medlock-Walton, "Thinking in blocks: Implications of using abstract syntax trees as the underlying program model," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 63–66.

[11] N. Tillmann, M. Moskal, J. De Halleux, M. Fahndrich, and S. Burckhardt, "Touchdevelop: app development on mobile devices," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 39.

[12] F. Devos and J. Verges, "Hacked," 2014. [Online]. Available: http://www.hackedapp.com/

[13] "Swift playgrounds," 2016. [Online]. Available: https://www.apple.com/swift/playgrounds/

[14] "Hopscotch," 2012. [Online]. Available: http://www.gethopscotch.com/

[15] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. scratch," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 346–350. [Online]. Available: http://doi.acm.org/10.1145/1734263.1734383

[16] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: Students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 199–208. [Online]. Available: http://doi.acm.org/10.1145/2771839.2771860

[17] A. Hidayat, "Esprima: Ecmascript parsing infrastructure for multipurpose analysis," 2012. [Online]. Available: http://esprima.org/

[18] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 257–281.