Automatic Programming and Education

Clayton Lewis
clayton.lewis@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

ABSTRACT

Automatic programming, as supported by recent language-model based AI systems, potentially allows a new approach to making computation a useful tool for learning, a goal of the Boxer project. This paper shows that the Codex system can be used to support some of the explorations in mathematics for which Boxer has been used. Virtually no knowledge of programming is required. Reflecting on the lessons from this exploration may sharpen the goals we bring to educational computing. What knowledge about computing, as distinct from the ability to creatively use computing, should learners gain?

CCS CONCEPTS

• **Social and professional topics** → **K-12 education**; • **Computing methodologies** → *Machine learning approaches*; • **Software and its engineering** → **Automatic programming**.

KEYWORDS

automatic programming, computational literacy, education, Boxer

ACM Reference Format:
Clayton Lewis. 2022. Automatic Programming and Education. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As Charles Rich and Richard Waters noted in a 1988 review ([6]), "automatic programming" is a goal as old as programming itself, for which expectations have changed radically as technology has advanced. Once, programming without needing to know assembly language would have been taken to be automatic, relative to the alternatives available, but higher level languages have long since pushed ambitions upwards. What Rich and Waters called the "cocktail party" version of the ambitions in 1988 was this:

> There will be no more programming. The end user, who only needs to know about the application domain, will write a brief requirement for what is wanted. The automatic programming system, which only needs to know about programming, will produce an efficient program satisfying the requirement.

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This was the "cocktail party" version, for Rich and Waters, because they felt it was uninformed, and unattainable. A key challenge was creating systems that could manage the extensive and very diverse knowledge that would be needed for this performance. Rich and Waters saw no prospect of this: "A further look into the future reveals no sign of the cocktail party version of automatic programming."

Knowledge representation in 1988 used symbolic structures and rules, constructed by hand. Recently, a very different approach has emerged in which large neural networks extract knowledge from enormous corpora, and structure it themselves, in ways that defy simple description. Because they create models of the use of language in their corpora, they are called language-model based systems. Their success has been remarkable. In particular, language-model based systems, trained on enormous bodies of program code, have made great strides towards realizing Rich and Waters' cocktail party vision. Can such systems contribute to the uses of programming that are envisioned in the Boxer project?

2 CODEX IS A LANGUAGE-MODEL BASED SYSTEM THAT GENERATES PROGRAM CODE.

Codex [2] is a neural net transformer system that is trained to predict the next token in sequences of tokens drawn from an enormous corpus of program code. The magnitude of the prediction task forces the system to develop very complex encodings of the input tokens, further complex codings of those codings, and so on, for several nested layers of encoders. Structures called attention heads, in each layer, enable the encoding of any entity to depend on other entities at the same level, so that dependencies between even widely separated entities can be detected.

The resulting encodings are very abstract, and not easily characterized. But they permit the system to predict the continuation of sequences of tokens that appear nowhere in the training corpus. It can do this because, when processing a novel sequence of tokens, it can respond to resemblances, in a very general sense, between the novel sequence and sequences that have been seen.

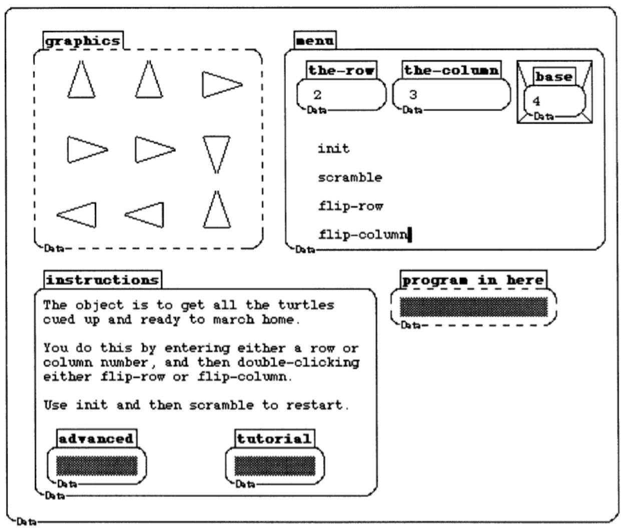
During training, a very large number of connection weights, about 12 billion, within Codex, are set to values that optimize prediction performance on its corpus. Once training is complete, these connection weights are never changed. In operation, Codex is given a new sequence, called a conditioner, or prompt, and its task always is to find a sequence of further tokens that is a probable continuation of the presented prompt. A probable sequence is output when a termination criterion is met.

The only information Codex receives about what it should do is the prompt, the sequence of tokens that it is asked to continue. In the explorations presented below, the prompts crucially include HTML comments, in English, that might appear at the start of a

Web page. Remarkably, Codex's corpus includes enough of this kind of text, associated with code, to enable it to follow the comments with appropriate code, in many cases. As will be illustrated below, this means that Codex can be used to create complete, working programs, with very little or no knowledge of programming needed by the user.

3 USING CODEX FOR EXPLORATORY PROGRAMMING IN A MATH TOPIC

Roschelle and Mason ([7]) and Mason ([5] report on a microworld, created using the Boxer system [4], that can be used to explore a family of problems that illustrate applications of group theory. The problems generalize the Three Cups problem, in which one is shown three inverted cups, and challenged to turn them all right side up, under the constraint that one must always turn over two cups on any move (the task is impossible). Roschelle and Mason explore much more complex situations, in which there is an array of objects, each of which can have a number of states, but legal moves have to change the state of groups of objects at once. For example, their system can represent a 3 x 3 array of stylized clocks, with hands able to point to noon, three o'clock, six o'clock, or nine o'clock, for which controls are offered that advance the hand of all the "clocks" in a row or column (see Figure 1 , from Roschelle and Mason, 1995, Figure 1.)



The object is to get all the turtles cued up and ready to march home.

You do this by entering either a row or column number, and then double-clicking either flip-row or flip-column.

Use init and then scramble to restart.

Figure 1: Boxer example from Roschelle and Mason, 1995.

Their system is intended to permit students not only to explore that form of the puzzle, but also to modify the puzzle, so as to have a different number of "clocks", or a different number of allowed positions for the hands.

Can this kind of program be created using Codex? The answer is yes, with some limitations. Figure 2 shows a screenshot of a Web application coded entirely by Codex. This program manages a grid of objects with two states, H and T, thought of as coins. This program allows one to flip the coins on the diagonals, as well as the rows and columns (as provided in Mason, 1995).

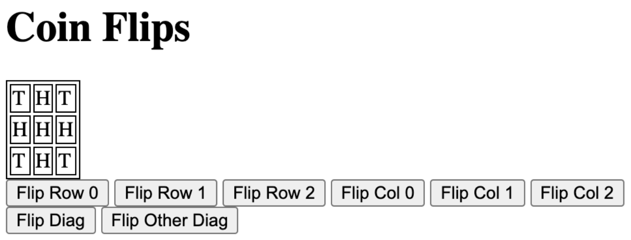


Figure 2: Screenshot of Coin Flips Program

Listing 1: Prompt for Coin Flip

```
<!-- Create a web page with the title "Coin Flips"
show a 4 by 4 grid of letters
make all the letters H
write a function flip that take H and returns T, and takes
    T and returns H
when a letter is clicked, flip it
write a function flipRow that takes a number and flips the
    letters in that row
provide a button for each row
when this button is clicked, apply flipRow to that row
write a function flipCol that takes a number and flips the
    letters in that col
provide a button for each col
when this button is clicked, apply flipCol to that col
add another button that flips the letters on the diagonal
add another button that flips the letters on the other
    diagonal
-->
<!DOCTYPE html>
```

The listing in Appendix A shows the code for this program, which is about 80 lines of CSS, HTML, and Javascript. Listing 1 shows the English language description provided to Codex, from which it generated the code.

The text shown in Listing 1 is not adequate, in that form, as a prompt for Codex. Rather, the text has to be provided incrementally, with the code produced at each stage being given back to Codex at the next stage.

To describe this process, let's continue to use "prompt" to refer to material that is given to Codex when asking it for code. We'll use the word "request" to refer to an English language description that is included in a prompt. A prompt may simply be a request, but more often, as we'll see, a prompt includes more than one request, together with code that Codex produced in response to one or more earlier prompts. We'll call the programs Codex produced at each stage "results".

Listings 2-6 describe the sequence of prompts that were used to obtain the code shown in the Appendix for the Coin Flip program. As seen in Listing 2,Prompt 0 consists of just a request, that we'll call Request 0. Codex produced the program shown as Result 0. The next prompt, Prompt 1, consists of Request 0, submitted again, followed by the code in Result 0, followed by Request 1, that asks

Codex to do a little more than Request 0. Prompt 2 is then formed from Request 1, the code in Result 1, and a slightly augmented Request 2. This pattern is repeated, until the result for the final, complete request has been obtained.

As can be seen, each request is presented in the form of an HTML comment, and followed by a tag that signals the start of a Web page. That framing implicitly lets Codex know that the desired result should be material that might follow the text in the request, that is, that the result should complete a Web page that might start with that comment.

Let’s consider some positive features of the example. First, a look at the code shows that a great deal of coding knowledge is implicit in this program, that is not present at all in the prompts. As user, I simply did not have to know any of it. What I say in the prompts is concerned almost entirely with what I wanted the program to do, and not with how my wishes have to be translated into code in some framework.(For clarity and concreteness, I will present my experience with Codex in the first person.)

Second, notice the buttons, and code, for operating on the diagonals of the grid. Knowing that Mason had included these operations, I wanted to include them too, but I was very uncertain that I could describe what I wanted, without getting into details about grid coordinates, and the like, that I feared might confuse the situation, as well as adding to what a user would need to know to create the program. I was surprised, and pleased, that the request shown in Prompt 3 was adequate for Codex to handle the main diagonal correctly.

Approaching the challenge of getting an operation on the other diagonal to work, I was actually quite sure that I would not be able to get Codex to accomplish this. Indeed, I would have to do some thinking myself to work out how to specify the other diagonal. For the main diagonal, row and column indices are equal, but what’s going on on the other diagonal? Fortunately, as can be seen, Codex spared me the need to worry about the matter.

On the less positive side, a little bit of computerese does show up in the prompts: the word function. While I can’t be sure that it would not be possible to get Codex to write a program like this without using that term, I can say that I did not succeed before introducing it. I believe the word function serves as a kind of a bridge between the operation associated with a button, and the button. That is, it may be easier for Codex to connect the code needed for a requested operation to a button if it is told to use the function construct, than if it has to work out the connection on its own. Here, anthropomorphic language stands in for real insight into how Codex’s behavior is organized, that we lack. I have collected some speculative thoughts on this in Appendix B

4 CLOCKS?

Now, what about a program like this that has clocks, perhaps like those in Figure 1, instead of just two-state coins (H vs T), as in the first example? I was able to create a program with n-state objects, as shown in the screenshot in Figure 3. Here the numbers represent the positions of hands on a grid of "clocks"; they advance from 0 to 3, and then return to 0. The final request for generating this program in in Listing 7. The sequential process by which the final request was realized was much like that for the Coin Flips example, save

Listing 2: Prompt 0 for Coin Flip, with Result 0

Prompt 0 consists only of **Request 0**:

```
<!-- Create a web page with the title "Coin Flips"
show a 3 by 3 grid of letters
make all the letters H
write a function flip that take H and returns T, and takes
    T and returns H
when a letter is clicked, flip it
-->
<!DOCTYPE html>
```

Result 0 is:

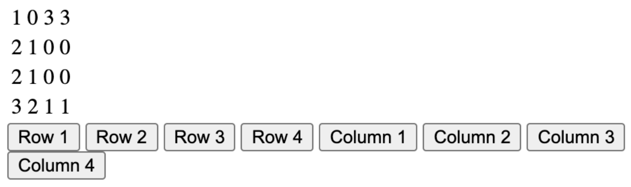
```
<!DOCTYPE html>
<html>
<head>
<title>Coin Flips</title>
<style>
table, th, td {
    border: 1px solid black;
}
</style>
</head>
<body>
<h1>Coin Flips</h1>
<table>
<tr>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
</tr>
<tr>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
</tr>
<tr>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
<td onclick="flip(this)">H</td>
</tr>
</table>
<script>
function flip(element) {
    if (element.innerHTML == "H"){
        element.innerHTML = "T";
    } else {
        element.innerHTML = "H";
    }
}
</script>
</body>
</html>
```

Listing 3: Prompt 1 for Coin Flip

```
Prompt 1 consists of Request 0, Result 0,
and Request 1, which is:
<!-- Create a web page with the title "Coin Flips"
show a 3 by 3 grid of letters
make all the letters H
write a function flip that take H and returns T, and takes
    T and returns H
when a letter is clicked, flip it
write a function flipRow that takes a number and flips the
    letters in that row
provide a button for each row
when this button is clicked, apply flipRow to that row
-->
<!DOCTYPE html>

Result 1 consists of the same code as Result 0, with the
    following code added at the end of the body:
function flipRow(row) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.rows[row].getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        flip(tds[i]);
    }
}
</script>
<button onclick="flipRow(0)">Flip Row 0</button>
<button onclick="flipRow(1)">Flip Row 1</button>
<button onclick="flipRow(2)">Flip Row 2</button>
```

that Request 0 asked only for the grid of numbers to be displayed, before asking for any operations on the numbers to be implemented. As shown, this program uses just numbers to represent the states



```
1 0 3 3
2 1 0 0
2 1 0 0
3 2 1 1
```

Figure 3: Numeric "Clocks".

of the objects, and not the more or less stylized clocks used in the Roschelle and Mason papers. That's because, despite a lot of effort, I was unable to get any form of graphical representation of the objects to work.

It's not that Codex can't "do graphics". It certainly can. I was able to get it to draw a single clock, and arrange for it to advance the hand of that clock, and I was able to get it to draw a grid of clocks, but not to advance the hands of groups of clocks in the necessary way. I believe the issue is the complexity of doing graphics on Web pages, as discussed further in Appendix B.

Layout is also a challenge. My efforts to get a 2D grid of clocks failed completely until I injected the word "table" into the prompts.

Listing 4: Prompt 2 for Coin Flip

```
Prompt 2 consists of Request 1, Result 1,
and Request 2, which is the same as
Request 1, with the addition of these lines:

write a function flipCol that takes a number and flips the
    letters in that col
provide a button for each col
when this button is clicked, apply flipCol to that col

Result 2 includes the code in Result 1, with the addition
    of following code, inserted in appropriate places, as
    seen in the final code listing in the Appendix.

function flipCol(col) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (i % 3 == col) {
            flip(tds[i]);
        }
    }
}
and
<button onclick="flipCol(0)">Flip Col 0</button>
<button onclick="flipCol(1)">Flip Col 1</button>
<button onclick="flipCol(2)">Flip Col 2</button>
```

Listing 5: Prompt 3 for Coin Flip

```
Prompt 3 consists of Request 2, Result 2,
and Request 3, which added this line to Request 2:

add another button that flips the letters on the diagonal

Result 3 added the following code to Result 2,
together with a button to activate it:

function flipDiag() {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (Math.floor(i / 3) == i % 3) {
            flip(tds[i]);
        }
    }
}
```

Without "table" as a hint, the code would draw the requisite number of clocks, but all in a horizontal or vertical line. Coordinating its "knowledge" of tables and graphics I conjecture is another part of the challenge for Codex.

While the example in Figure 3 falls short of replicating some of the desirable features of the Roschelle and Mason programs,

Listing 6: Prompt 4 for Coin Flip

```
Prompt 4 was constructed in the same way from Request 3,
and Result 3, adding this additional line to Request 3
to form Request 4:

add another button that flips the letters on the other
        diagonal

Result 4 is the code for the finished program, as shown in
        the Appendix.
```

Listing 7: Final Request for Clock

```
<!-- Create a web page with the title "numbers"
show a 4 by 4 grid of numbers
write a function advance that takes a number and returns
        the number plus 1, modulo 3
write a function advanceRow that takes a number and
        advances the numbers in that row
provide a button for each row
when this button is clicked, apply advanceRow to that row
write a function advanceColumn that takes a number and
        advances the numbers in that column
provide a button for each column
when this button is clicked, apply advanceColumn to that
        column
-->
<!DOCTYPE html>
```

there are some bright spots. The program does support much of the exploration functionality Roschelle and Mason want. In fact, the version shown here was created from a version in which the grid was 3 x 3, by using the prompt shown in Listing 8. Similarly, using a prompt in which "modulo 4" is changed to (say) "modulo 5", in the second request, gives a program in which the "clocks" have five possible states. The grid size for the Coin Flip program can also be changed, in the same way.

Changing the grid illustrates the peculiar character of working with Codex. Having been surprised by the ease with which Codex handled the diagonals, in the first example, I was not surprised that it could cope with a change from 3 x 3 to 4 x 4. However, it proved unable to make the change from 3 x 3 to 3 x 2! It generated code that draws only a 2 x 2 grid, whether I asked for 3 x 2 or 2 x 3.

That last point, the unexpected failure to handle certain grid sizes, illustrates a larger, pervasive issue. It isn't possible to know, in advance, what Codex will do correctly, and not. Further, code can look correct, but not work correctly. This is a practical irritation even for simple problems, and would be a huge drawback for larger problems, for which extensive testing would be needed to know that one had really gotten what one asked for.

Relatedly, sequences of prompts described for the example programs mask the fact that a good deal of trial and error was needed to find prompts that actually gave the desired effect. The use of

Listing 8: Prompt to Change Grid for Clock Program

```
<!-- Create a web page with the title "numbers"
show a 3 by 3 grid of numbers
write a function advance that takes a number and returns
        the number plus 1, modulo 3
write a function advanceRow that takes a number and
        advances the numbers in that row
provide a button for each row
when this button is clicked, apply advanceRow to that row
write a function advanceColumn that takes a number and
        advances the numbers in that column
provide a button for each column
when this button is clicked, apply advanceColumn to that
        column
-->
<!DOCTYPE html>
 ...
--code for the request above included here--
 ...
</html>
<!-- Create a web page with the title "numbers"
show a 4 by 4 grid of numbers
write a function advance that takes a number and returns
        the number plus 1, modulo 3
write a function advanceRow that takes a number and
        advances the numbers in that row
provide a button for each row
when this button is clicked, apply advanceRow to that row
write a function advanceColumn that takes a number and
        advances the numbers in that column
provide a button for each column
when this button is clicked, apply advanceColumn to that
        column
-->
<!DOCTYPE html>
```

"function", mentioned earlier, is an example; it took some time to find that using that term was important to Codex's success.

Further, because Codex is stochastic in some of its operations, repeating the same prompt will usually give different results. This means that if a prompt does not produce working code, it can be worthwhile submitting the same prompt again, hoping for better luck.

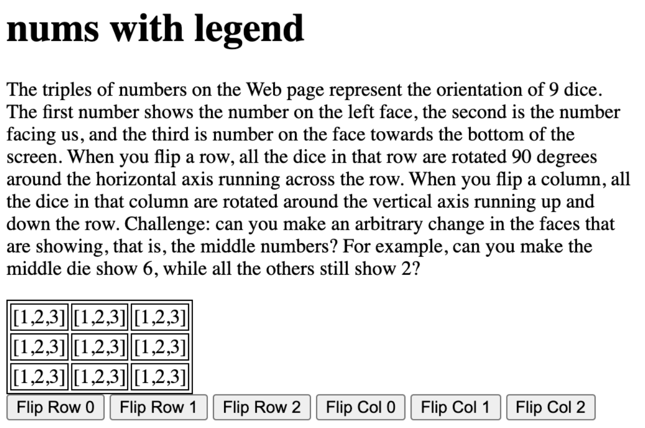
This random behavior gives rise to the worry that one might get bad code in response to pieces of prompt for which one had received good code earlier. But an effect of giving code back to Codex in a later prompt seems to be to freeze a correct response into the later code.

5 ROTATING CUBES

To further evaluate Codex's potential as a tool for exploratory computing, I used it to investigate a problem that is posed in [5], but not pursued in the paper:

> A cube is placed on each square of a chessboard. The faces of the cubes are congruent to the squares of the board. Each of the cubes has at least one black face. We are allowed to rotate a row or column of cubes about its axis. Prove that by using these operations, we can always arrange the cubes so that the entire top side is black.

As described above, I found that any Web graphics, let alone 3D, would be hopeless with Codex, so I didn't attempt to draw the cubes. Instead I contented myself with a numerical representation, as shown in Figure 4, a screenshot of the finished Web program. The program can be used online as a Web page at https://claytonhalllewis.github.io/cubes/, where the code can be viewed with an appropriate selection in a browser. The final request in the prompt sequence for the program is shown in Listing 9. The triples of numbers on the page represent the orientation of 9 dice. The first number shows the number on the left face, the second is the number facing us, and the third is the number on the face towards the bottom of the screen. When you flip a row, all the dice in that row are rotated 90 degrees around the horizontal axis running across the row. When you flip a column, all the dice in that column are rotated around the vertical axis running up and down the row.



nums with legend

The triples of numbers on the Web page represent the orientation of 9 dice. The first number shows the number on the left face, the second is the number facing us, and the third is number on the face towards the bottom of the screen. When you flip a row, all the dice in that row are rotated 90 degrees around the horizontal axis running across the row. When you flip a column, all the dice in that column are rotated around the vertical axis running up and down the row. Challenge: can you make an arbitrary change in the faces that are showing, that is, the middle numbers? For example, can you make the middle die show 6, while all the others still show 2?

[1,2,3]	[1,2,3]	[1,2,3]
[1,2,3]	[1,2,3]	[1,2,3]
[1,2,3]	[1,2,3]	[1,2,3]

Figure 4: Numerical Presentation of Rotating Cubes.

The development of this program had some unexpected twists. I planned to use just two numbers to represent a die, the number on the left face, and the one showing on the top. The column rotation is pretty straightforward in that representation: the configuration (m, n) goes to (n,7-m), given the way dice are numbered, in which opposite faces add to 7. But the row rotations seemed pretty murky, so I made up a partial table of inputs and outputs for that rotation, and set to work to get Codex to do what I wanted.

Providing it with the code I had from the coins, that is quite similar in structure, it wasn't too hard to get it to do the basic layout, but the rotation operation was a problem I thought Codex would just write a bunch of tests (basically a switch on the input), but it wanted to be cleverer, and produced a mess. Once I tumbled to that, I gave it heavier hints, things like "look up the input pair in this list of inputs, and return the corresponding element in this list of outputs", but, though it "tried hard", it couldn't get it. One issue is that the Javascript indexOf method uses a comparison under which arrays with the same values don't show as equal. It certainly looked as if the code, using that method, was fine; it just didn't actually work, because the array elements were themselves arrays.

Listing 9: Final Request for Rotating Cubes Program

```
<!-- Create a web page with the title "nums with legend"
put this text on the page: "The triples of numbers on the
        Web page represent the orientation of 9 dice. The
         first number ... REMAINDER OF LEGEND TEXT HERE...
        example, can you make the middle die show 6, while all the
        others still show 2?"
make a 3 by 3 grid of triples of numbers with [1,2,3] in
        each triple
write a function rotH that takes a triple [a,b,c]
and computes d = 7-c, then returns a new triple [a,d,b]
when a triple is clicked, apply rotH to it
write a function rotHRow that takes a number and applies
        rotH to the triples in that row
provide a button for each row
when this button is clicked, apply rotHRow to that row
write a function rotV that takes a triple [a,b,c]
and computes d = 7-b, then returns a new triple [d,a,c]
write a function rotVColumn that takes a number and applies
         rotV to the triples in that column
provide a button for each column
when this button is clicked, apply rotHRow to that column
-->
<!DOCTYPE html>
```

That forced me to think harder about representation. I realized that by using 3 numbers, rather than 2, both rotations are made easy. That is, if the triple p,q,r represents a die with face p to the left, face q on top, and face r pointing down the page,

```
Row rotation of p,q,r gives p,(7-r),q
Column rotation of p,q,r gives (7-q),p,r
```

The reason it's so much easier is interesting: two numbers suffice to specify the position of a die, if and only if you know its handedness. There are two distinct ways to number a die, given the constraint that opposite sides add to 7. With a standard die, if you look at the corner where faces 1,2, and 3, meet, the numbers increase clockwise. On a die with the opposite handedness, the numbers increase counterclockwise. The results of rotating the two kinds of dice are different, and the rotation code has to reflect this.

The three number representation uniquely specifies not only the orientation of the die, but also what handedness it is. For example, a standard die can be in position (1,2,3), but a nonstandard one can't.

This example raises a distinction about representations, to which I will return below. In the earlier examples, Codex allowed me to ignore nearly all questions of representation. Arguably, representation is an essential aspect of computation, one that provides much of its value (see http://comprep.blogspot.com/). So an educational practice in which representation can be ignored can't be good. But the rotating cubes example shows that the situation is more complex. Working with Codex allowed me to ignore a great many aspects of representation, such as how triples are represented as arrays; how these are laid out the screen, in HTML; how click handlers are specified, and much else. But it enabled me to focus on other, arguably more important questions of representation, in particular, how to represent the orientation of a cube, and rotations of it. That is, the example suggests that Codex allowed me to ignore intellectually uninteresting representational issues, while attending to interesting ones.

The example raises another point that is relevant to a comparison with Boxer, in particular, as a computational tool. I wanted to add a description of the triples representation, and an example problem, to the Web page. Initially I failed completely to get Codex to add such content. In Boxer this sort of thing is utterly trivial: you just type what you want, where you want it, so annotations, hints, explanations, and suggestions for things to try are very simple to add. One could say that Codex is the polar opposite of a direct manipulation system. one can never work directly on the results of a program to adjust what the program does.

Eventually I returned to the task, and succeeded. The key was to get Codex to create code for a page titled "legend", with just the legend text, and then give a prompt that combined the prompt and code for that, with the prompt and code for the "nums" page, that contained the program without the legend, and a request that combined the two.

An interesting wrinkle was that I needed to change the title for the combined page from just "nums" to "nums with legend". Earlier attempts, just like this, only without the change of title, had failed. One might think that things like page titles could not matter "to a computer", but with Codex, they do matter.

6 DISCUSSION

With these experiments as background, how would working with Codex measure up to Andy diSessa's goals for Boxer, creating a computational literacy? Here is a statement from diSessa [3]:

> To oversimplify, there are two contrasting conceptions of a computationally enhanced literacy. The dominant one today I would describe as a trivial literacy. In this view, experts and software designers will supply the general populace with highly tuned and elegant tools and other pieces of software that we will learn to use in the niches for which they were intended-symbol manipulators, graphers, simulations and simulation tool kits, and the like.
>
> In contrast, a deep computational literacy offers one crucial additional resource- in-principle access for everyone to the creation and modification of the dynamic and interactive characteristics of the medium, the very characteristics that define the medium as an extension and improvement of text in the first place. In metaphorical terms, reading without writing is only half a literacy. Deep computational literacy means "writing" in addition to "reading," creating as well as using.

Would working with Codex provide a literacy, in this sense? In discussing this question, we'll often take Boxer as kind of foil, to highlight the potential strengths and weaknesses of Codex in this role. Let us approach this by reviewing lessons from the experiments.

6.1 Codex allows someone to create programs of the sort suggested for intellectual explorations in math class, as illustrated by the examples from Roschelle and Mason, with very little knowledge of programming.

This has been demonstrated in the examples above. Note that the claim is limited to this kind of program, where behavior is simple and easy to check, and efficiency and strong confidence in correctness aren't important. For other kinds of programs knowledge of programming would be needed.

6.2 Codex allows one to modify these programs, in ways Roschelle and Mason suggest, still with very little knowledge of programming.

This has also been demonstrated, with limitations. It is possible to change grid size for programs, and the number of states of the objects being manipulated, as Roschelle and Mason ask. But not all changes were successful.

As noted, changing the *shape* of the grid did not work, on the initial attempt. It is possible that a more concerted effort, involving getting Codex to change the grid out of context, and then merging that solution in with the earlier program, would work. This approach did work for the legend problem for Rotating Cubes.

That success remains uncertain brings out a difference between Codex, as a tool for exploration, and Boxer. With Codex, one has no way to know whether a given way of asking for something will work, other than by trying it. With Boxer, once one has made the investment of learning the relevant concepts, one can feel fairly confident about one's results. If one's understanding of Boxer suggests that something should work, then it likely will work.

6.3 Using Codex requires understanding problem decomposition.

Getting Codex to work depends on breaking a problem into parts, that are individually easy enough for Codex to handle, and that can be combined into a complete solution. Arguably, the ability to decompose problems is one of the skills one would like to see, in the budget of intellectual benefits that a computational medium would provide. Learning mathematics can also develop this skill, but perhaps not as concretely as computing does.

It seems that Codex actually relies more on this skill than conventional programming, including programming in Boxer. While teachers of programming stress the importance of modular design, and Boxer provides good support for it, modularity is not required to make things work. Students very often neglect it, even after they begin working on problems large enough to really punish them for their neglect. With Codex, you can't really get anywhere without breaking even a simple problem into parts.

A little more than just breaking a problem into parts seems to be needed. As discussed, without the hint provided by the programming term "function", Codex seemed to have difficulty organizing solutions to the sample problems.

As usual, it is uncertain how firm the requirement is for the use of this term, or whether other terms might also work. But even supposing that this word is needed, one can suggest that having to get a sense of what kinds of operations can be described as functions is a valuable part of the skill of problem decomposition.

6.4 In working with Codex, a great many uninteresting representational details can be ignored, while more fundamental ones can be focused on.

This point was illustrated most clearly in the Rotating Cubes example. As a Codex user I did not have to think at all about how to lay out the number triples on the screen, or the difference between an array and a list, or how to refer to a particular element of the grid of triples, or much else. I did have to think about how to use numbers to represent the orientations of cubes, and the interesting issues there had to be, and could be, explored.

In evaluating programming languages, one can use a notion of *payload ratio*. Starting with a computational idea, one writes a program that implements it, and then goes through the code, noting what statements, or parts of statements, express parts of the computational idea, that is, the payload, and how much of the code is needed just to make the program work, that is, overhead. For many programming languages, declarations fall in the latter category, for example. The payload ratio is just the rough proportion: how much of the code expresses the computational idea, and how much does not, but has to be written anyway?

From this standpoint, Codex looks very good. There's very little in the prompts that isn't in the computational idea. Had I written the code for the Web page myself, a great deal of what I wrote would have been overhead, overhead that Codex spared me.

Tedious though it was to work with Codex, I nevertheless felt that it provided me with a good computational scratch pad for Rotating Cubes. I was free to wrestle with the key challenge of representing the rotations, and really did not have to worry about all the other things that went into the working program.

6.5 Codex allows one to create programs that are integrated into external computational ecosystems; in the examples, the Web.

A persistent issue in programming reform is what is sometimes called the *walled garden* problem. One creates an attractive computational world, in which one feels that one can do things in a way that one really likes, elegantly clear and intelligible. Then one faces the need to connect one's creations to the big outside world, producing results in a form that other systems can consume, and consuming results from them. Often the simplicity and niceness of life inside the garden is shattered, as outside considerations can't be kept out.

Codex is interesting in this respect, in that it consumes expressions framed in the walled garden of the user's conception, but creates code that lives in the world outside. The code Codex created for me included CSS, HML, and Javascript, all that cruft that has been patched together to make the Web. But I did not have to be concerned with any of that, to get a working program that anyone can run in their browser. The OpenAI Codex Live Demo at https://www.youtube.com/watch?v=SGUCcjHTmGY shows that Codex can be used to interface with a specified API, and gives further examples of its ability to connect the garden to the world.

Whether this arrangement is wholly a good thing or not in a computational medium depends on one's goals. One goal could be to have such a capacious garden that one never needs to go outside, to get one's intellectual work done. Learning "real programming" would then be a separate venture, undertaken only by those who (for some reason) want to work with computation outside the garden, say professional software engineers.

Codex would not be a good choice for the capacious garden goal. Why? Because in the end Codex is working with the characteristics of the code in its corpus, that is, "real programming" code, and not the reformed code in one's garden. Boxer, as a representative vision of an inclusive garden, is a better fit here.

What if, instead, one's goal is for one's computational medium to help one adjust to the big world, in the sense of being a step towards "real programming"? Boxer may seem stronger than Codex in this respect, in that there is some overlap between what a Boxer user has to think about, and what a "real programmer" has to think about. "Real programmers" have to keep track of different kinds of things, and what can be done with them, like different data types. Boxer offers a simple set of distinctions– there aren't many different kinds of boxes– but there are differences. In Codex, on the other hand, one needn't deal with these differences among things, explicitly, at all, as we've seen in the examples. Boxer has control structures, too, and Codex, as used in the examples, has them only implicitly: they are there in the code, but hardly at all in what the user has to say.

It may be, though, that one could use Codex in a way that does involve understanding and dealing with these issues. Indeed, one use that is envisioned for Codex is as a programmer's assistant: the user is a programmer, and writes conventional code, and just gets help from Codex with the details [2]. Thus Codex might form part of a more or less conventional programming curriculum.

Finally, what if one's goal is to inhabit a garden in which things are simple, and little specialized knowledge is required, but at the same time wants one's creations to be able to connect with things on the outside? As we've seen, Codex does seem to be able to fill that bill, at least in the example of creating ordinary Web applications. The OpenAI Codex Live Demo, mentioned earlier, shows that much more can be done.

Boxer could be extended to interface with any given external system, presumably, but developer effort would be needed to do that. Codex would need work, too, to create explanations for students of how to use a given platform, and this work, too, would likely need to be done by a programmer, who could understand the target system well enough. But it wouldn't be software development work.

6.6 Codex has important functional limitations, including handling graphics, and layout.

As mentioned, I was unable to create graphical representations of clocks that would work. The issue seems to be that there are many ways to create graphics on the Web, and Codex has trouble making appropriate choices, based on the operations one needs to perform. For example, sometimes Codex would propose code that would display a canned image of a clock, which is hopeless for making the hands move.

It is plausible that this limitation could be eased by creating a collection of examples that could be included in one's prompts, when one needs graphics. Conceivably a form of turtle graphics could be supported in that way, but this is far from clear.

Layout is also problematic, and success would probably require knowing more about HTML than would be ideal, and giving Codex a lot of hints based on that knowledge. Here the contrast with Boxer is stark: there one just puts things where one wants them.

6.7 Codex, in its current form, is frustrating to work with.

Using naked Codex, as I did, where one provides a prompt, and collects the output, is tedious in many ways. Here are some.

One can't tell whether the code will work without trying it out, which means saving the code somewhere, and running it.

Often a prompt will not give working code on a first attempt, and one has to choose between asking for another (stochastic) generation for the same prompt, that may end up working, or making a change to the prompt.

One can't tell whether a given prompt will work or not. it often took several iterations to find prompts, and a sequence of prompts, that would work.

Assembling a prompt, so that it includes appropriate earlier prompts, and associated code that Codex has generated for those prompts, and an appropriate new request, is fussy.

Some of these issues could be helped by fairly simple tooling, and indeed such tooling quite likely exists. For example, a tool that would automatically bring one's code up in a browser would be great.

6.8 Codex is the opposite of concrete.

One of the design goals for Boxer is to make computation concrete. For example, the values of variables can simply be exposed to view, and program execution can be visualized in detail.

Codex could not be more different. Unless one can read code, as a hypothetical math student exploring clocks cannot, one sees only one's own description, and the running program. Nothing about program execution is exposed, and variables can be seen only if one has asked to see them. Programs contain some variables one doesn't even know are there.

But does this matter, and in what way? For me, it was liberating to be able to explore the Rotating Cubes problem, as a working Web application, without having to bother with computational details. Some of the same questions about goals that came up when discussing the walled garden problem are relevant here. Perhaps I don't need to have computation made concrete to do what I want with it, and perhaps I do, for other purposes.

6.9 Like Boxer, Codex is not one of those "highly tuned and elegant tools and other pieces of software that we will learn to use in the niches for which they were intended."

A common paradigm for computing in the schools is that students are given tools created for specific purposes: a genetics simulator, or a system for manipulating diagrams in geometry. One drawback of this approach is that students have to learn to use to use many tools, rather than a single framework, as with Boxer. A possibly larger problem is that students, and teachers, can only explore paths that have been identified and paved by the tool creators. They can't go off in directions of their own choosing. In particular, even if the tools they are given have broad functionality, students won't able to work easily across the domains of coverage of different tools, as Antranig Basman (personal communication, January 19, 2022) has pointed out. It is widely recognized that domain boundaries like these need to crossed, but that there are strong forces that act to build them up (see Donald Campbell, [1]. It will be good if computational tools can help reduce the barriers, rather than strengthen them.

In this respect, Codex is like Boxer. There's no a priori scoping of what the tool can and cannot be used to explore (though there are practical limitations, at least for now, as already discussed.)

The late Mike Eisenberg, who was a participant in the community at MIT from which Boxer arose, argued forcibly and often for the importance of supporting students' interests. As long as educational research is focused on finding educational interventions for broad classes of learners, things that work pretty well for most learners, it will be bounded away from truly realizing its potential. That's because extraordinary impact, for a learner, comes from following their own intellectual direction, not one imposed on them as members of a broad class.

7 CONCLUSION

Codex, and similar language-model based tools, have taken large steps towards the goals of automatic programming researchers of the 1980's. As a medium for computational exploration in school, it has intriguing capabilities. Whether one is attracted to its unusual potential, or not, thinking about it may contribute to clarifying our goals, as the educational computing community reflects on the inspiring intellectual legacy of the Boxer project, and how its ideals can be realized.

ACKNOWLEDGMENTS

I thank Antranig Basman, Katherine Hermann, Owen Lewis, and Jason White for many useful discussions.

REFERENCES

[1] Donald T Campbell. 2017. Ethnocentrism of disciplines and the fish-scale model of omniscience. In *Interdisciplinary relationships in the social sciences*. Routledge, 328–348.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[3] Andrea A DiSessa. 2001. *Changing minds: Computers, learning, and literacy.* Mit Press.

[4] Andrea A. diSessa and Harold Abelson. 1986. Boxer: A reconstructible computational medium. *Commun. ACM* 29, 9 (1986), 859–868.
[5] John Mason. 1995. Exploring the sketch metaphor for presenting mathematics using boxer. In *Computers and Exploratory Learning*. Springer, 383–398.
[6] Charles Rich and Richard C. Waters. 1988. Automatic programming: Myths and prospects. *Computer* 21, 8 (1988), 40–51.
[7] Jeremy Roschelle and John Mason. 1995. Sketching a Multidisciplinary Microworld: A Collaborative Exploration in Boxer. In *Computers and Exploratory Learning*. Springer, 289–304.
[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

A CODE FOR COIN FLIP EXAMPLE

```
<!DOCTYPE html>
<html>
<head>
< title >Coin Flips</ title >
<style>
table , th, td {
    border: 1px solid black;
}
</style>
</head>
<body>
<h1>Coin Flips</h1>
<table>
<tr>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
</tr>
<tr>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
</tr>
<tr>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
<td onclick=" flip ( this ) ">H</td>
</tr>
</table>
<script>
function flip (element) {
    if (element.innerHTML == "H") {
        element.innerHTML = "T";
    } else {
        element.innerHTML = "H";
    }
}
function flipRow(row) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table .rows[row].getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        flip (tds[ i ]) ;
    }
}
function flipCol (col) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (i % 3 == col) {
            flip (tds[ i ]) ;
        }
    }
}
function flipDiag () {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (Math.floor(i / 3) == i % 3) {
            flip (tds[ i ]) ;
        }
    }
}
function flipOtherDiag () {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (Math.floor(i / 3) == 2 - i % 3) {
            flip (tds[ i ]) ;
        }
    }
}
</script>
<button onclick="flipRow(0)">Flip Row 0</button>
<button onclick="flipRow(1)">Flip Row 1</button>
<button onclick="flipRow(2)">Flip Row 2</button>
<button onclick="flipCol(0) ">Flip Col 0</button>
<button onclick="flipCol(1) ">Flip Col 1</button>
<button onclick="flipCol(2) ">Flip Col 2</button>
<button onclick="flipDiag() ">Flip Diag</button>
<button onclick="flipOtherDiag() ">Flip Other Diag</button>
</body>
</html>
```

B MORE ABOUT CODEX

I include here material not needed in discussing Codex as a computational tool for education, that may be of interest to those who are interested in the system itself.

Codex is a transformer [8], a complex, multilayer neural network whose internal functioning is poorly understood. Elhage, Olah, and coauthors at https://transformer-circuits.pub/2021/framework/index.html make some progress in unpicking some aspects of information flow in these systems. They suggest that some of the functioning of the network is best understood as copying material from the prompt to the output. The way the prompt and output are represented internally actually means that nothing can literally be copied, but rather must in a sense be reconstructed. Nevertheless, behaviorally, much of what the system does has the effect of copying. One can see this in the examples of the prompts and associated code, where stretches of the output code are identical to stretches of the prompt.

In other cases it seems that the system is copying material from its corpus, that is, from the vast body of code on which it was trained. For example, Codex produced code for Prompt 0 of the Coin Flips example, but that prompt included no code at all.

Here, too, actual copying is simply not possible. The system has access to the corpus only during training, and not at all during operation. And even during training, it is only given stretches of code from the corpus to practice on, so to speak, and never has access to the entire corpus at once. Nevertheless, we'll speak about "copying", from the prompt and from the corpus, as if that is what is happening.

All of the examples involve re-presenting to Codex code that it itself generated, in response to an earlier prompt. One can speculate that this helps by giving Codex more relevant code to copy. The importance of this suggests, plausibly, that "copying" from the prompt is easier than "copying" from the corpus.

Perhaps relatedly, the importance of problem decomposition seems to be that Codex has extra difficulty "copying" material from the corpus, when the request to be satisfied is complex. If it succeeds in finding material for a simple request, and that material is then re-presented in a prompt, it can work more effectively by "copying" from the prompt, rather than from the corpus.

What's involved for both sources, corpus and prompt, is more than simple "copying". The code needs to be adapted, and in just the right way. For example, in the Rotating Cubes program, the code for rotating columns,

```
function rotVColumn(col) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        if (i % 3 == col) {
            rotV(tds[i]);
        }
    }
}
```

is very similar to that for rotating rows, that was provided in the prompt:

```
function rotHRow(row) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.rows[row].getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
        rotH(tds[i]);
    }
}
```

But, as can be seen, it is not identical. The "copy" process has to work out when the material can just be "copied", and when it has to be adapted, and how.

In this case the transformation looks like a simple analogy, simply involving the replacement of "row" by "col", and "rotH" by "rotV" but that is misleading. The prompt fragments corresponding to these two stretches of code are

"write a function flipCol that takes a number and flips the letters in that col" ,

in the earlier part of the prompt, and

"write a function rotVColumn that takes a number and applies rotV to the triples in that column"

in the current request. The correspondences here are not trivial, involving identifying "flips the letters" with "applies rotV to the triples".

In the same example, compare the code

```
function flip(element) {
    if (element.innerHTML == "H") {
        element.innerHTML = "T";
    } else {
        element.innerHTML = "H";
    }
}
```

in the example provided in the prompt, with the corresponding code

```
function rotH(element) {
    var table = document.getElementsByTagName("table")[0];
    var tds = table.rows[0].getElementsByTagName("td");
    var td = element.innerHTML;
    var split = td.split(",");
    var a = parseInt(split[0].substring(1));
    var b = parseInt(split[1]);
    var c = parseInt(split[2].substring(0, split[2].length
        -1));
    var d = 7-c;
    var e = [a,d,b];
    element.innerHTML = "[" + e[0] + "," + e[1] + "," + e
        [2] + "]";
}
```

generated in response to the request to rotate the triples. There is the same high level structure in both, including the use of innerHTML to get and set the values. But the bodies of the two functions are quite different. The code for splitting up a triple, and putting it back together, had to be "copied" from the corpus, not from the prompt. It's also noteworthy that the prompt said nothing at all about how to perform the spitting and reassembly, or even that these operations would be needed.

It appears that the "copying" process is affected by a kind of interference. The system may be able to generate correct code for request A, and for request B, but not for A and B requested at the same time. But it likely will be able to satisfy A and B, if it is given its separate solutions for A and B to work from, when given the combined request.

Other contextual effects show that Codex's response to input is quite different from what one expects from "computers". On the negative side, asking for a Web page with title "Clocks" often brought in irrelevant, and quite complex, code dealing with clocks, even though the code was not responsive to what was actually requested, functionally. That is, even though one would normally think, "This title is just a word, it doesn't affect what 'the computer' will do, Codex is affected by it. On the positive side of this effect, choosing the page title "nums with legend" helped Codex integrate code for displaying the legend into the code for the nums functionality, as mentioned earlier. Here, "helped" is anthropomorphized language for "It didn't generate the correct code until I changed the page title."

Difficulty in "copying" from the corpus may be responsible for Codex's challenges with graphics, as discussed in the body of the paper. Because there are different ways to create graphics on the Web (<img> tags, canvas, svg), and all of these are represented in its corpus, Codex has to decide what to "copy". It is easy to get this wrong, because some forms of graphics don't readily support some of the operations that others do. For example, the transform attribute of svg makes rotation easy, but if Codex has chosen a different form of graphics, things won't work out well, if rotation turns out to be needed. It's hard to help it over this kind of problem, because, as mentioned earlier, it doesn't respond well if asked to do too many things at once. So one can't just tell it everything it will need to do, in an effort to help it make the correct choices.

To insert an education point here, it seems likely that human learners face the same challenge. If there are many ways to do something, selecting an approach, as a response to a simple cue, isn't possible. This is an educational argument for Boxer's approach of providing a single, integrated medium, over the alternative of a mosaic of specialized tools for different problem domains.