

Large Language Models and the Psychology of Programming

Clayton Lewis

University of Colorado Boulder

clayton.lewis@colorado.edu

Abstract

Large Language Models have emerged rapidly as powerful coding tools, in some cases showing the ability to create entire working programs, and (more commonly) providing help with the details of a great many APIs and frameworks. This emergence raises a number of questions for the PPIG community. Will these tools change what programmers do, in ways that affect "the psychology of programming"? Given the (apparent) command that these systems have of natural language, and of the semantics of a great many domains of activity, can they be used to enhance the kinds of interactions with software tools that PPIG researchers have studied? Noting that large language models exhibit analogical reasoning as an emergent capability, can they leverage insights from early research on programming by analogy? Does predictive modelling, as a key cognitive process that can be applied in many domains, suggest new ways to think about programming not based on text?

1. A new approach to programming

1.1. Domains and mapping

[P]rogramming is often defined as a process of transforming a mental plan that is in familiar terms into one that is compatible with the computer ... many [bugs] and difficulties arise because the distance between these is too large. This concept is called closeness of mapping by Green & Petre ... The closer the programming world is to the problem world, the easier the problem-solving ought to be. Conventional textual languages are a long way from that goal.

There are also many ambiguities in natural language that are resolved by humans through shared context and cooperative conversation Novices attempt to enter into a human-like discourse with the computer, but programming languages systematically violate human conversational maxims because the computer cannot infer from context or enter into a clarification dialog. (Pane, Myers, et al., 2001)

As programmers we have been used to mapping entities and structures in a *problem domain* into a *representational domain* that includes things like variables, lists, statements, and so on. The psychology of programming has been the psychology of this mapping process. Aspects of it have included how to learn about and understand the representational domain, how to form the mapping itself, and how to evaluate the effectiveness of the mapping, for example in debugging.

The mapping is multifaceted, with concerns in different facets often being to some extent separable. For example, we may separate the development of a "high level" view of a program, from the details of elaborating this view as a syntactically correct expression in a given programming language.

2. Large Language Models (LLMs) change this picture dramatically.

Figures 1, 2, and 3 show screenshots of programs created using the LLM GPT-4. Each figure also shows the initial input provided to GPT-4. Hereafter I will refer to the system simply as GPT.

The example in Figure 1 is based on an example developed originally in Boxer (diSessa & Abelson, 1986; Roschelle & Mason, 1995), a system created to support the development of a deep computational literacy in schools. As shown in the figure, the initial prompt for that example was:

I'd like the code for a Web page that shows a 3 x 3 grid of H's and T's, Each row and column should have a button that flips all the H's in the row or column to T's, and vice versa.

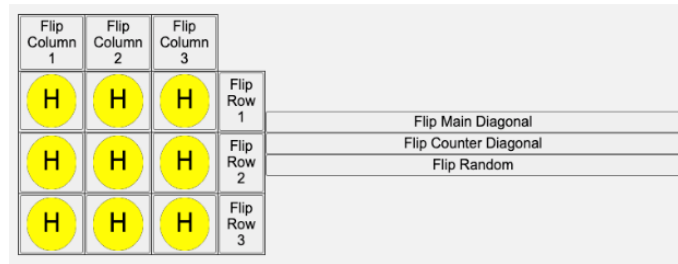


Figure 1 – Initial prompt– I'd like the code for a Web page that shows a 3 x 3 grid of H's and T's, Each row and column should have a button that flips all the H's in the row or column to T's, and vice versa.

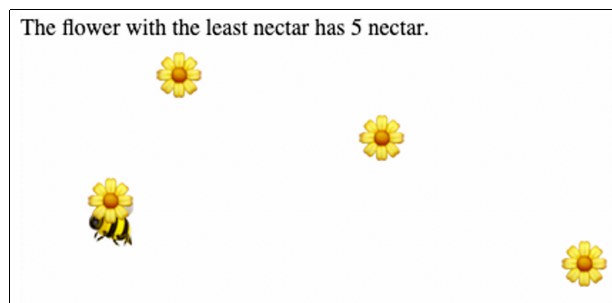


Figure 2 – Initial prompt– Make a Web page that shows a bee and several flowers. Each flower has a certain amount on nectar in it. Make the bee visit all the flowers, looking at how much nectar they have, and show which flower has the least nectar.

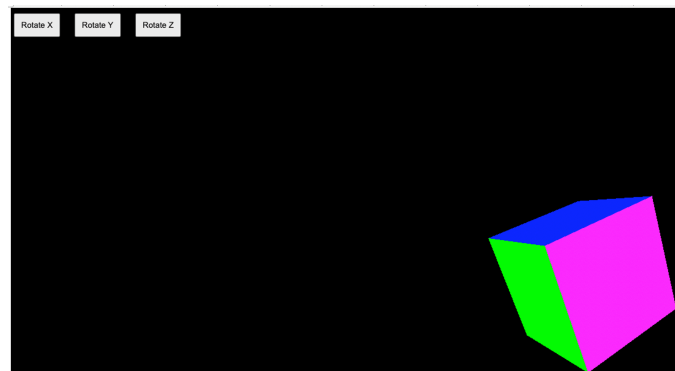


Figure 3 – Initial prompt– Can you give me a Web page, all in one file, that draws a wire frame cube, and has buttons to rotate the cube around the x,y, and z axes?

The program shown in Figure 1 was developed in stages, with the following additional inputs.

Can you put that all in one file for me? [The system had initially provided separate HTML, CSS, and Javascript code files.]

Actually, I do want those buttons for the rows and columns. Can you put those in? [The system had produced code for these buttons, but not the buttons themselves, and said that it had left the buttons out (no idea why, as it had no problem adding them).]

Can you add buttons to flip the two diagonals?

Can I have a button that will flip all the coins randomly?

Now, can you replace the H's and T's with pictures of coins?

That works fine, but I'd rather not have to use those img files. Can you change the program so that it draws the coins itself? [The system had explained where to use img files with the program, and that had worked fine, but finding suitable images had required some work.]

The versions of the program created at each stage worked correctly.

Examining the inputs that produced these results, we can see that the user (me) had no need to know anything about the syntax or semantics of HTML, CSS, or Javascript, or even to look at the code. I did need to know how to run the Web page from my browser, though had I not known that, I could have asked the system:

User:

If I have a Web page in my computer, how can I run it?

GPT:

To view a web page that is stored locally on your computer, you just need a web browser such as Google Chrome, Firefox, or Safari. Here's how you can do it:

Find your HTML file: This file should have a .html or .htm extension. This is the file that your browser can interpret and display as a web page.

Double-click the HTML file: In most operating systems, double-clicking the HTML file will automatically open it in your default web browser.

If double-clicking doesn't work for some reason, or if you want to open the file in a different web browser, you can do so manually:

Similarly, I knew enough about directories to be able to put the img files in the right place, when using those, but I could have asked the system about that, too.

The example in Figure 2 is based on the tutorial for John Pane's Hands system (Pane & Myers, 2006), which was designed to support natural programming. Pane and collaborators did a great deal of research to explore what features of a programming language would make it easy for nonprogrammers to use it.

In the example, a bee and some flowers are shown on the screen, with each flower having some amount of nectar. The bee visits each flower, and then displays the minimum amount of nectar found in any flower.

This example was more difficult for GPT than the coins example. The initial input,

Make a Web page that shows a bee and several flowers. Each flower has a certain amount of nectar in it. Make the bee visit all the flowers, looking at how much nectar they have, and show which flower has the least nectar.

produced a page with the flowers lined up in a column. Later versions corrected that, but failed to show the bee moving. It took a total of six inputs to get the working version. It's important to note, though, that none of the inputs required any knowledge of the code, or potential problems in it. Rather, typical inputs were

The flowers and text are now fine, but the bee does not seem to be moving to the flowers.

or

I'm still not seeing the bee move, unfortunately!

These inputs, and all the others, are framed in the problem domain, not the representation domain.

By contrast, the original Hands tutorial (Pane & Myers, 2006) on which the example is based, includes more than ten pages of conceptual material that users must understand, for example this:

The front of the card has spaces on it; this is where we can add information. Each row on the card is a property. It has a pair of boxes to hold information: the boxes on the left are called name boxes; the boxes on the right are called the value boxes. Soon you will see what these names and values are for.

Handy gives each new card three properties to start with. We can see the names of these properties by looking at the name boxes. Can you see the names of the three properties that we are starting with? The properties are named: cardname, x, and y. Now let's find out what these properties mean.

Look at this card's first property, cardname. The value box for the cardname property has the name of the card, Card-1. Since we haven't named our flower yet, Handy has given it this name to start with. Handy also shows the card's name at the very top of the card. Do you see it?

Besides cards and properties, the tutorial introduces lists, events, and rules, and the admonition, in bold text, that Handy will only understand rules written a certain way.

In its historical context, of course, Hands represented a notable advance. Hands is quite easy to understand, compared to

```
languages = ['Swift', 'Python', 'Go', 'JavaScript']
# run a loop for each item of the list
for language in languages:
    print(language)
```

or

```
char c;
bool isLowercaseVowel, isUppercaseVowel;
cout << "Enter an alphabet: ";
cin >> c;
isLowercaseVowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
isUppercaseVowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');
if (!isalpha(c))
    printf("Error! Non-alphabetic character.");
else if (isLowercaseVowel || isUppercaseVowel)
    cout << c << " is a vowel.";
else
    cout << c << " is a consonant.";
```

let alone

```
global _start

        section .text
_start: mov     rax, 1           ; system call for write
        mov     rdi, 1           ; file handle 1 is stdout
        mov     rsi, message ; address of string to output
        mov     rdx, 13          ; number of bytes
        syscall
```

The challenge and opportunity of LLMs is that people can now create programs without understanding the representational domain of computational concepts at all. No syntax, no semantics of computational entities, not even any higher level view of the program. In short, many of the the traditional concerns of the psychology of programming, as we have conceived it, or at least what we have seen as the central concerns, simply vanish, for tasks like these.

3. Analogy and programming

The concept of analogy appears often in the literature of the psychology of programming. Often it arises as an aspect of how people understand programs and programming, when an analogy is used to connect something in the domain of computation to something familiar. For example, Ahadi, Lister, and Teague (2014) report the use of "box" as an aid to understanding variables (and they note the misleading suggestion that, like boxes, variables ordinarily can contain multiple values.)

(Is this use of the "box" concept an analogy, or a metaphor? it could be either, and I won't take up the difference here. The more an explanation is offered of the relationship between two things, the more the relationship is an analogy (A. F. Blackwell, 1996).

Do such relationships play any role in how LLMs create programs? Superficially, it might appear so. GPT is certainly able to explain the box analogy, if asked to do so; see Box 1. Note that it mentions the difficulty with the box model that Ahadi et al. reported. It can also *propose* an analogy to explain variables, if asked to do so. However, in one interaction, it suggested different vegetables in a recipe, and the possibility of replacing one vegetable by another, as useful things to think about, in understanding variables. That seems quite odd, and indeed GPT itself suggests that a container might be better, at the end of the vegetable response.

Box 1

```
User: is a variable in a programming language like a box?
GPT: Yes, that's one way to understand it!

A variable in a programming language can be thought of as a box where you can
store a value. Just like you might label a box with what's inside (like "
holiday decorations"), variables are given names to indicate what data they're
storing.

When you create a variable in a programming language, you're essentially telling
the computer: "Hey, I need a box where I can keep this specific piece of
information." And when you assign a value to that variable, you're putting
something into that box.

For example, you might have a variable called age and assign it the value 25. In
this case, age is the label on the box, and 25 is what's inside the box.

Later on, when you use that variable in your code, the computer will go to the "
box" and see what's inside. If you change what's inside the box - that is, if
you change the value of the variable - the computer will use the new value
next time it checks.

It's worth noting that this box metaphor has its limitations. For example, unlike
physical boxes, a variable can only hold one value at a time; assigning a new
value to a variable replaces the old value. Moreover, different types of
variables (like integers, strings, or arrays) can hold different types of data
, whereas a box could hold a mix of different items.

And in languages that include reference types (like objects in JavaScript or
classes in Python), the variable might hold a reference to the value rather
than the value itself - more like a note saying where the box is, rather than
the box itself. But despite these limitations, the box metaphor is a helpful
way to start understanding what a variable is and how it works.
```

More generally, analogical reasoning is a strength of LLMs. Webb, Holyoak, and Lu (2023) show that GPT-3 performs well on a wide range of tasks used to assess or study analogical reasoning in humans.

Despite these facts, it's doubtful that connections between computational structures and things like boxes are involved when LLMs create programs. Rather, connections between programs and problem statements seem to be important.

An example reported by Lewis (2021) can illustrate this. An LLM has likely encountered in its corpus many programs that involve looking things up in collections, and producing an appropriate message, if an item is not found. But it is unlikely that it has ever seen a program that keeps a list of gifts that someone has given, and the recipients, as was called for in the example. Yet in the program the LLM created, the message "No gifts found for Fred" was used, if Fred was the recipient in the search. By some means, the LLM has adapted the messages seen in its corpus of example code to the particular task at hand.

That description, with its use of the word "adapted", should be taken only as saying *what* the LLM has done, and not *how* it has done it. The nature of its training, and its internal operation, mean that the LLM rarely has access to any stored representation of the examples it has seen, that it could "adapt". Rather, the LLM has created a model that relates requests for code to code. This model produces an appropriate message, as well as the rest of the code that the prompt calls for.

The examples presented earlier illustrate this same process. The LLM's model is able to map a wide range of requests for code to code that is responsive to the requests.

Past efforts at automatic or technically aided programming have sometimes focussed on analogical relationships between programs. Brand's Programming without Code proposal, at PPIG 2016 (Brand, 2016), built on Anderson and Thompson's PUPS model (Anderson & Thompson, 1989), in which analogical transformations of programs are supported by *annotations*.

Consider this code for factorial:

```
factorial(x) {
    if (x==0)
        return 1;
    return x*factorial(x-1)
}
```

The analogy summorial : factorial :: + : * can be solved by replacing * by + in the code, after annotating the 1 in the base case by "identity for *" and noting that "identity for +" is 0.

Dershowitz (Dershowitz, 1986) shows how a whole family of rather diverse programs can be developed, by implementing analogical relations between members of the family. Starting with a program that uses binary search over an interval to approximate the quotient of two numbers, Dershowitz derives a program that uses binary search to approximate the cube root, and then one that finds an element in an ordered list. Each step of the process can be framed as solving an analogy: what program is to quotient, as this given program is to cube root?

GPT can reproduce a progression of programs of this kind, in response to these requests:

```
Here is some code for finding the quotient of two reals. Please modify it so it
computes the cube root of a number, instead.
```

and

```
Based on that, can you give me a program that takes an array of numbers, in order,
and a number that occurs in the array, and returns the index of that number
in the array?
```

The main difficulty in this is that GPT does not think that binary search is a good way to compute quotients, so getting the starting program takes a little persistence. Later on, on the list problem, it strayed from the form of the example, but readily responded appropriately when asked

```
That works, but can you make the code closer to the example I gave?
```

While GPT can produce a progression of programs that share a common algorithmic theme, as Dershowitz envisioned, its operation is quite different from that described in Dershowitz's account. Der-

showitz's code is heavy with annotations, like the following:

$$\text{achieve } |a^{\frac{1}{3}} - r| < e, \text{ varying } r$$

or

$$\text{assert } A[z + s + 1] \leq b < A[z + 1]$$

These express the effect or purpose of the code, and developing a new version of a program, to serve a modified purpose, calls for close reasoning about how the annotations need to be transformed, when the problem is changed.

Notably, GPT needs none of this apparatus. Its model is able respond directly to a modified program requirement, accompanied by code to be adapted. to create the desired transformed program.

Arguably Dershowitz's annotations, while not needed by GPT to transform the code, could be useful in avoiding errors. When done correctly, reasoning about the annotations, and the accompanying code, could detect possible coding errors. How well GPT might be able to reason about program annotations is a subject for further investigation. Box 2 shows a very simple example.

Box 2

```
User: evaluate this pseudocode:
x=3
y=4
x=y
y=x
assert x=4
assert y=3

GPT: The pseudocode you've provided appears to have a mistake. Here is the line-by-
-line explanation of the pseudocode:

x is set to 3.
y is set to 4.
x is set to y, which means x is now 4.
y is set to x, which means y is now 4 as well, not 3.
So, both x and y are equal to 4, not 3. Therefore, the final two lines of the
pseudocode, where you assert that x is 4 and y is 3, will fail for y because y
is not 3, it's 4.

If you intend to swap the values of x and y, you need to use a temporary variable,
as shown below... In this corrected version, x is indeed 4 and y is indeed 3,
so the assertions will pass.
```

4. Conversation

A progressive strand in work at PPIG (and elsewhere (Van Brummelen, Weng, Lin, & Yeo, 2020)) is the exploration of conversational interactions with software tools. Church, Söderberg, and McCabe (2021) review many ways in which human conversation supports effective communication, and envision ways in which these benefits might be secured in (for example) exploring errors in code found by a compiler.

That GPT supports such interaction has already been suggested in the examples above. We've seen that the user can rely on GPT to maintain the context needed to interpret an input like "Can you add buttons to flip the two diagonals," in which "add to what?" and "what two diagonals" need not be spelled out.

Similarly, the input "I'm still not seeing the bee move, unfortunately!" makes no reference to the code at all, but is correctly interpreted as a request to examine the code, and find and fix in it whatever is responsible for the indicated shortcoming.

Box 3

User: Can you give me a Web page, all in one file, that draws a wire frame cube, and has buttons to rotate the cube around the x,y, and z axes?

GPT: Sure! You can use HTML to structure the page, CSS for styling, and JavaScript with the Three.js library to generate and control the 3D cube. Here's a basic example of how this could work:

Please note that JavaScript is disabled in this text interface, so you will need to copy and paste the code into an .html file and run it in your browser to see it in action.

[code given here]

In this example, we first include the Three.js library with a script tag. Then we set up a basic scene with a camera and a renderer. We also create a cube with a wireframe material and add it to the scene. Then we set up an animation loop that will render the scene.

We also set up three buttons that, when clicked, will rotate the cube around the X, Y, or Z axis respectively.

Finally, we set up an event listener that will resize the canvas if the window size changes.

This is a very basic example. Depending on your needs, you may want to add more functionality, such as different types of controls or other 3D objects.

User (after viewing the page): That's pretty good, but the axes change when I rotate the cube. I'd like the axes to be fixed when the cube rotates.

GPT: If you want the rotation buttons to always rotate the cube relative to the original axes, not the current orientation of the cube, then you would need to change how the rotations are performed.

Here is a modification of the previous script that implements this. It works by creating three different rotation matrices for the X, Y, and Z rotations, and then applying these to the cube's position:

[code given here]

In this version of the script, when a button is clicked, the corresponding rotation matrix is applied to the cube's transformation matrix, causing it to rotate relative to the world axes, rather than its own axes. This gives the effect of the axes remaining fixed as the cube rotates.

User: Could you color the faces of the cube with different colors?

GPT: Three.js does not natively support different colors for each face of a BoxGeometry directly in one Mesh. But you can create six different Meshes each with a PlaneGeometry, color them differently, and group them together to form a cube.

Here's an example of how you can achieve this:

[code given here]

This script creates six planes, colors them differently, positions and rotates them to form a cube, and then groups them together using a THREE.Group object. Then it applies the rotations to the group, so all the planes will rotate together.

[code works as desired.]

Box 3 shows the example for Figure 3, including not only the turns taken by the user, but also the responses from the system. One can see that GPT has no trouble placing the user's inputs in the right context, and understanding them, including the somewhat tricky matter of how the axes should be handled. One can also see that GPT assumes that the user wants to know quite a bit about how the code works. In a tutorial interaction, as has been common in this kind of situation, that assumption is reasonable. In fact, though, since I no longer need to understand the computational representation, I actually did not want these explanations! (I likely could have asked GPT to omit this material, but I haven't tried that. It's not impossible that suppressing the explanations would actually damage GPT's performance, since its outputs form part of the input when processing later requests.)

Another interaction, not presented here in detail, illustrates some additional conversational capabilities. I wanted to create a client for accessing the GPT models via their API. Starting with the input

```
I need a Web page that will allow me to get continuations from the OpenAI davinci
model. Can you give me the page, and tell me how to set it up?
```

I obtained a working client (actually, a server-client pair). GPT handled some intermediate requests, like "Can you tell me how to set that up using node.js?" and "Great. Can you tell me more about how to get my API key?" Its explanations were adequate for me, whose knowledge of UNIX-style commands and practices is sketchy at best, to do what was needed.

After completing the setup as directed, there was trouble. GPT responded appropriately to the inputs:

```
After starting the server I get the message 'Cannot GET' at localhost:3000

That seems to be working, but I just see [object Object] on the page, instead of
the completion,

that's giving me 'Cannot read properties of undefined (reading '0')'

now I am getting 'AxiosError: Request failed with status code 404' and no console
output.
```

It provided guidance that I could act on, except in the last case, when it listed seven things to check, most of which I felt unable to pursue in an effective way.

That was the only situation, among all those discussed here, in which I looked at the code, other than to insert code snippets where GPT directed. I noticed that whereas I had asked for a client for a model called "davinci", the code mentioned "davinci-codex". That superficial difference triggered my next input:

```
User: Is it possible that "https://api.openai.com/v1/engines/davinci-codex/
completions" in my server.js should have just "davinci" rather than "davinci-
codex"?

GPT: Yes, it's definitely possible. ...

So, to answer your question, if you want to use the Davinci model, you should
indeed change "davinci-codex" to "davinci" in your URL.
```

After dealing with one more error, for which GPT's guidance was easy to follow, the application worked as desired. Even including the one case of examining the code, just mentioned, GPT allowed me to accomplish this with no real understanding of the code concerned.

In the process, GPT fielded a mix of inputs, including ones that called for more explanation of what to do, so as to support a novice user, and ones that reported errors that needed to be corrected. The inputs could be framed in English in a natural way, with no attention to what GPT would understand or not understand, just as would be true with a knowledgeable human conversational partner. There was no need to mark the difference between inputs of different kinds.

GPT's guidance was usually provided in a clear and actionable way, except for the list of issues to check, mentioned near the end of the interaction. Code changes were described by providing new code,

in complete form, together with a clear indication of where in the program the new code needed to be placed.

In a few cases, especially near the start of the interaction, GPT's guidance assumed knowledge I did not have. But requests for more guidance were simple to pose, and were responded to effectively.

5. Understanding the problem domain as well as the computational domain

In mapping problem statements or requests to code, GPT can draw not only on its knowledge of programming, but also on its knowledge of many aspects of the world and human affairs. Here the word "knowledge" perhaps should carry scare quotes, since the relationships among what GPT "knows" how it can be said to "know" it, and human knowledge and knowing, are unclear. But the role of this "knowledge" in the following examples will be apparent.

The first example begins with the following request:

```
User: I need a Web page that lets people enter reports of vandalism in their
      neighborhood, and sends an email with their report to vandalism@gmail.com.
```

This would just be silly as an input to a conventional programming tool. A compiler or interactive development environment can't be expected to know anything at all about vandalism. GPT provides code for a page with fields for name, address, and description. This further request

```
User: That's ok, but I thought maybe you could suggest (for example) some common
      kinds of incident, to make it easier for someone to make a report.
```

leads to a form with a drop down menu for five categories of incident, graffiti, property damage, littering, illegal dumping, and other, with working Javascript code that implements this simple system. In its responses one can see that GPT "knows" quite a bit about vandalism, and what would be involved in reporting it.

This is a very new situation in the realm of programming. Commonly people who understand programming don't understand a target domain in which they may need to work, and people who understand the target domain don't understand programming. Indeed, Curtis, Krasner, and Iscoe (1988) cited "the thin spread of application domain knowledge" among software developers as a leading cause of system development failures.

In a study of software design organized by Petre and Van Der Hoek (2013), skilled, professional designers were challenged to develop a system for simulating traffic flow, in a situation that included traffic signals with traffic sensors. None of the designers knew about, or sought out, representations that have been developed in this domain for problems of this kind, nor about common problems that arise, such as the *yellow trap*. The "yellow trap" is a specific issue in traffic engineering where a driver is "trapped" in the dilemma zone during the yellow phase of a signal. This usually occurs in a situation where left-turning drivers mistakenly believe that oncoming traffic is also facing a yellow light, which can lead to collisions if the oncoming traffic actually has a green signal. But GPT does know about the yellow trap; indeed the definition just given was provided by it.

Further, GPT knows about useful representations for such problems, such as

```
Signal Sequence Charts: This is a more detailed type of timing diagram that not
      only shows when each signal is red, yellow, or green, but also when each
      vehicle movement (straight, left, right) is permitted. This would clearly show
      that left turns are only allowed during the protected left turn phase.
```

That description is part of GPT's response to the request "I'm worried about 'yellow trap'. what representation might help with that?"

But can GPT coordinate this problem domain knowledge with its knowledge of the domain of computation, as it was able to do in the much simpler vandalism example? The answer is a qualified yes. The request

Could you give me the HTML and js for a Web page that would allow me to simulate a yellow trap, and a control scheme that avoids it?

led eventually to a working simulator, shown in Figure 4.

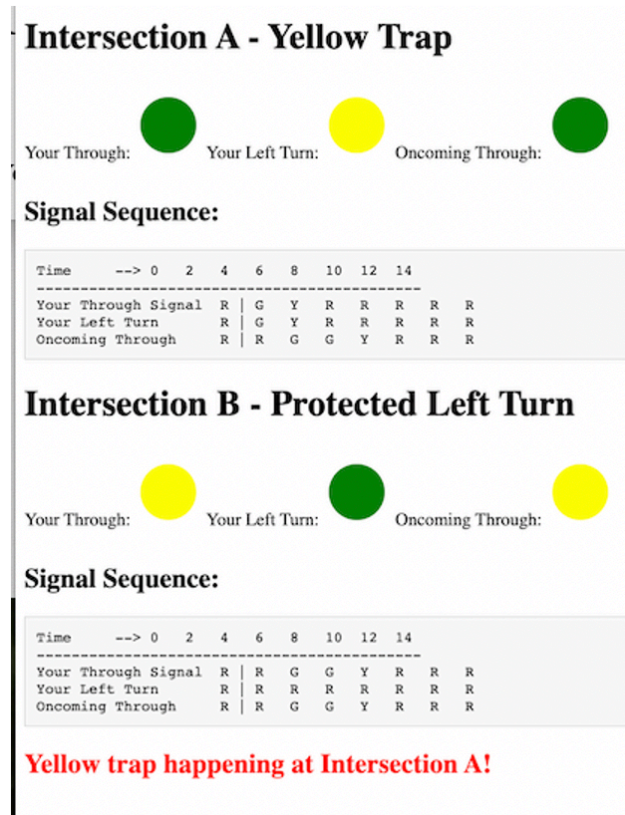


Figure 4 – Traffic signal simulator

This page is animated, with the traffic signals changing color so as to show that yellow trap occurs for Intersection A, with the "Your left turn" signal being yellow while the "Oncoming Through" signal is green. That does not happen in Intersection B.

The qualification of the positive assessment stems from two issues. First, the interaction required 12 follow-up requests to produce the result shown. Some of the follow-ups were refinements, such as slowing the simulation to make it easier to follow, or giving a specific alert on the screen when a yellow trap occurs. But seven of the follow-ups were needed to report that the simulation was not showing yellow traps at all, that is, that the code did not satisfy the requirements. GPT was successful in correcting the problems, with no guidance about the code from the user.

However, the value of GPT's problem domain knowledge is undercut by the occurrence of these failures. While one might hope that working with GPT would mean that a user could learn about yellow traps from the simulation that they asked GPT to create, the user needed to know what a yellow trap is, and to recognize that it was not happening, in order for GPT to create a working simulator. GPT's "knowledge" on its own was not sufficient.

Two of the follow-on requests elaborated on another aspect of the original request, that the system should simulate a "control scheme". Specifically, a request was issued that the system should use and show a "signal sequence chart", one of the representations it had itself suggested as appropriate for understanding the yellow trap.

Unfortunately, the representation it uses, as shown in the figure, is not actually a "signal control chart",

as used by practitioners. Indeed, the representation in the figure does not fit the description GPT itself provided for these charts, quoted earlier. Such a chart "not only shows when each signal is red, yellow, or green, but also when each vehicle movement (straight, left, right) is permitted." The representation in the figure does not show that. Further, the representation does not reflect what the simulation actually does.

These limitations aside, we can see that GPT indeed "knows" a good deal about traffic signals, and is able to create code that usefully represents much of that "knowledge". For example, it "knows" that traffic signals have green, yellow, and red aspects, that these aspects cycle in the indicated order, and that there are signals for traffic in two directions, in situations where a yellow trap can occur, that there can be signals for left turns, and so on.

The examples in the sections above show that GPT offers ways to program that are quite different from familiar programming tools and environments. Users can describe their intentions, without being concerned with the code that will implement them. But the expression of intentions must be text.

There are many visual programming systems that use diagrammatic representations, rather than text. These are much discussed at PPIG and elsewhere (see among many, Good and Brna (1999); Rosian (2022)). But these provide different ways to express intentions in the *computational domain*, such as rules or data paths, not the *problem domain*. LLMs, or similar systems, might support different modes of expression.

6. Does the new approach to programming offer new opportunities for programming not based on text?

While GPT is trained on a corpus of text, related developments work with non textual material, such as movements or images (Brohan et al., 2022; Khan et al., 2022). Do these suggest that there may be ways to create programs that do not require textual expression of any kind?

It might seem that *no-code systems*, like Google Appsheet, already offer this. Such systems replace coding by selections in a family of dialog boxes (Appsheet requires writing some expressions, too.) These systems don't use textual expression, and they allow intentions to be expressed without concern for code (though as in visual programming, one's intentions have to be translated from the problem domain into a computational domain, in the case of Appsheet a domain of tables and views.) Unfortunately these systems are full of hidden dependencies, and hidden information of other kinds. The consequences of one's selections are invisible, and only become manifest when the application one is building runs or fails to run.

Could the capabilities shown in systems like GPT help? No-code systems seem weak as the basis for models like GPT, even if a corpus of actions (button clicks, form filling in the no-code tool) and effects (Web pages and interactions with them) could be gathered. The same hidden effects that make no-code systems challenging for human users might present problems for training systems like GPT, too. But this could be an interesting problem to explore. Very preliminary attempts, using manual coding of commands and effects, as used in Lewis (Lewis, 1988), show that GPT can learn to interpret commands from a few examples. These actions and effects are far simpler than those seen in Appsheet, though.

6.1. Multimodal systems based on predictive modeling might support new kinds of nontextual programming

The core of an LLM is *predictive modeling*: given a sequence of tokens in the corpus, what comes next? A complex neural network is trained to make accurate predictions in that setting. The resulting model can be used to make predictions that continue any sequence of tokens, including sequences that occur nowhere in the corpus. Remarkably, this predictive model, adapted by a limited amount of "fine tuning" using examples of human interaction, is able to perform the tasks we've been discussing.

In principle, the same approach could be used to make predictions for corpora of other kinds, for example corpora of gestures. As mentioned, there are systems that operate over domains of this general kind, like

robot movements (Brohan et al., 2022). Perhaps more usefully for present purposes, corpora might mix gestures and code. Humans operate habitually in a mixed domain of this kind, that includes gestures and speech (Kendon, 1994). In ways we are in general unaware of, gestures help our words convey meaning.

Conjecturally, we function in this mixed domain by identifying *regularities*, that is, patterns with predictive value, that contain elements of both kinds, so that a sequence of words can predict a gesture, or a sequence of words and gestures can predict a word, and so on. Further, it appears that the learning process results in the establishment of analogies between regularities. For example, a relationship between a sequence of gestures and a sequence of words could be extended to a relationship between those gestures and a different, but related, sequence of words.

Analogies between regularities can be identified with some kinds of (often implicit) concepts. For example, the concept of Ohm's p-prim (phenomenological primitive), identified by DiSessa (1993), expresses the analogy between

more voltage and same resistance gives more current

and

more push and same friction gives more motion

and many other situations, in which magnitudes of various kinds are related as those in the examples are. The concept applies when there is an input of some kind, and an output, promoted by the input, and some intervening factor that modulates the effect of the input. Given this very general concept, a gesture that might accompany the assertion about electricity could also be used to help express the assertion about push and friction. Could similar relationships, between gestures and code, or between gestures and program behavior, be useful in programming?

The question is bafflingly broad. A narrower question, perhaps more approachable, is, could these relationships make programming more like sketching? A joint meeting of PPIG and the Art Workers Guild (London, September 5-7, 2018; see report at <https://ppigattheartworkersguild2018.wordpress.com/>), included testimonies from Guild members that cast programming, as repellent, in the literal sense, an activity that people would like to avoid, even if it is useful. "Sketching" in the question stands for another kind of activity, lacking these repellent qualities, and having the attractive qualities of enjoyable expression. As Charlie Gere asked at that meeting, can programming be like sketching?

Lewis (2019) presented some fragmentary suggestions about this, in which lines and ribbons have defined behaviors, that direct the movement of dynamic structures called vines. Figure 4 shows an example of a sketch that forms palindromes. Could a predictive model do anything to make this programming paradigm more useful, while retaining its sketch-like character?

This does seem barely possible. An idea that might be explored posits a model trained on encodings of the drawing actions that create the sketches, along with discourse about the resulting programs, and their behaviors. Could this model supply interpretations of drawing actions not included in the training corpus, in the same way GPT can continue sequences of tokens that do not occur in its corpus? These interpretations could then be supplied to an LLM, that would produce conventional code that implements them.

The chain of activities would be:

1. user enacts drawing actions
2. predictive model trained on actions and discourse, based on example sketches, generates discourse appropriate to the drawing
3. LLM generates code from the discourse

Given such a system, it might further be possible to intermix discourse about the intended purpose of a drawing, with the drawing actions. Doing this could make it more likely that the discourse generated in

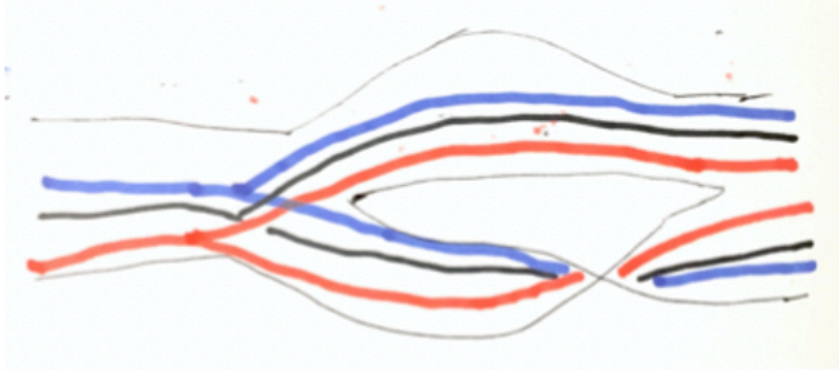


Figure 5 – A sketch that creates palindromes.

Step 2 would be more appropriate to the drawing, and its creator's intent. Thus the code generated in Step 3 could be more appropriate.

Such a system might address a key limitation of the proposal in Lewis (2019), that there is a fixed repertoire of drawing actions, with conventional interpretations. Not only does this impose definite constraints on what can be expressed, but also it imposes a substantial burden on the user, who must learn these actions and their interpretation. In the speculative system, users could draw whatever they like, and hope to get a satisfying effect, at least some of the time.

The system would support a wide range of sketch-likeness. At one extreme, users could draw freehand, and be satisfied with the resulting behavior. As they devoted more time to looking at example sketches and their effect, and as they said more about their intentions, the system would become less like sketching, and somewhat more like programming as we have known it. Even in this less sketch-like usage, however, users would not be concerned with the bit, bytes, lists, and variables that contribute at least some of the repulsion that the Art Workers feel.

7. What is ahead for the psychology of programming?

7.1. Artificial and natural semantics

What is different about using a predictive model like GPT to create programs, from programming as we have known it? Summarizing, users can express what they want in the problem domain, not the computational domain; they can describe their intentions iteratively, in conversation; the tool "knows" a good deal about many problem domains.

The conversational interaction that ties these differences together is supported by semantics for what users say that is quite different from the semantics of programming languages. The key feature of *artificial semantics*, as seen in programming languages, is that the meaning of expressions is determined by the expression, by applying specified rules of interpretation. In *natural semantics*, as seen in human language, and in interactions with LLMs, the form of an expression contributes something to the meaning assigned to it, but the influence of the form is often dominated by a complex context.

In artificial semantics, an expression means what the rules of the system say it means. The rules of Fortran, for example, say what a Fortran statement means.

In natural semantics, as Garfinkel (1969) observed, an utterance means what it has to mean, in order for it to have meaning, in the context in which it is used. For example, if I were to say, in September, 2023, "I really enjoyed giving my talk at PPIG in Stockholm," "Stockholm" means Lund. It would be understood to mean that by any interlocutor who knows the context.

The character of natural semantics isn't expressed only when repair is needed. The examples presented

earlier, like human discourse, show many cases in which expressions aren't wrong, but can only be interpreted in the light of context. For example, as we saw, in "Can you add buttons to flip the two diagonals," the questions "add to what?" and "what two diagonals" did not have to be answered. They are taken to mean what they need to mean for the request to make sense, in its context.

Historically, efforts at "natural language programming" like REL English (Henisz-Dostert & Thompson, 1969), or natural programming, like Hands, have stayed on the artificial side of the boundary. While REL looked like English, its interpretation was governed by rules. Similarly, the extended Hands tutorial was needed to enable users to know enough about how Hands expressions are interpreted to accomplish their goals.

The ribbons and vines system in Lewis (Lewis, 2019) also uses artificial semantics. As discussed, users would have to learn the acceptable drawing forms and their conventional interpretation. The speculative extended drawing system uses natural semantics to assign meaning to drawings, even novel ones.

The ability to support natural semantics in an LLM like GPT comes from the nature of its predictive model, and how it is trained. The model can exploit any regularities in its corpus, that is, any patterns that have predictive value, and use these in making predictions in new situations. Because the training corpus intermingles discourse about all kinds of things, as well as a wide variety of code, predictive regularities that link problem domain material to code can be found, and are used.

One of the first interactions with an LLM that captured my attention illustrates this blending of considerations. I wanted to include descriptive text on a Web page much like that shown in Figure 1 (see Lewis (2022)). But the Codex LLM did not add the text, that was provided to it, until I changed the title of the requested Web page from "nums" to "nums with legend". In a system with artificial semantics, it just doesn't matter what title is assigned to something. But in natural semantics, it *does* matter. Indeed, it's the role of titles in the natural semantics of humans that explains why titles actually exist!

Further perspective on the distinction between natural and artificial semantics comes from reflecting on the intentional programming project of Simonyi, Christerson, and Clifford (2006). Recognizing the challenge of coordinating knowledge of problem domains with knowledge of coding, they say:

For the creation of any software, two kinds of contributions need to be combined even though they are not at all similar: those of the domain providing the problem statement and those of software engineering providing the implementation. They need to be woven together to form the program.

The problem statements would be expressed in domain-specific languages:

Were the domain chess, for example, we would need a def for the pieces, the colors, the board and its squares, and the various states: initial, check, checkmate, draw. It is not required that we define the game in the schema - presumably that will be a part of the program that we create - but we need to have the vocabulary complete enough to define the game.

Creating such languages would be a colossal enterprise. But the problems go well beyond the effort involved. Phil Agre observes that deployment of information technology in work settings imposes, implicitly or explicitly, a grammar of action:

Grammars of action frequently oversimplify the activities they are intended to represent, if only because the people who articulate the grammars are only superficially acquainted with its actual complexities and the actual social forces that determine its form The ontology may fail to make enough distinctions, or else whole subcategories of "invisible" activity might go unrepresented. The grammar might impose overly restrictive ordering constraints

on the unitary actions, it might neglect the interleaving of distinct forms of activity, or it might mistake prescribed procedures for an accurate descriptive account (or at least a practicable form) of the activity As a result, the participants in the newly instrumented activity will find it necessary to evolve a system of "work-arounds" to keep things going. (Agre, 1994)

More broadly, and long ago, Harold Garfinkel argued that human activities, including communication, simply cannot be accounted for by the kinds of rules and structures that support artificial semantics.

Garfinkel approaches the topic by stressing that understanding language is not to be regarded as a matter of 'cracking a code' which contains a set of pre-established descriptive terms combined, by the rules of grammar, to yield sentence meanings which express propositions about the world. ... An utterance is thus the starting point for a complicated process of interpretative inference rather than something which can be treated as self-subsistently intelligible. (Heritage, 2013, p 139)

We can suggest that the workings of LLMs are much closer to Garfinkel's idea of meaning than are the forms of artificial semantics seen in programming languages, including domain specific languages.

7.2. Cognitive dimensions and natural semantics

A very productive line of development in the psychology of programming has been cognitive dimensions analysis (Green, 1989; A. Blackwell & Green, 2003), more fully, analysis of the cognitive dimensions of notations. A cognitive dimension is an aspect of a notational system that is likely to cause problems for, or assist, a user who must create, interpret, or modify the notation needed in some situation. For example, *hidden dependency* describes a situation in which two or more aspects of a notation are linked, so that one aspect can't be changed without changing another, or can't be interpreted without understanding another, but the connection between the aspects is not apparent. One can anticipate that a hidden dependency will be a common source of errors and frustrations for users. More generally, *viscosity* names a complex of considerations that make it difficult to make changes in the notation for something. Even dependencies that are clearly apparent can create viscosity.

To the extent that computerized tools continue to use notations, cognitive dimensions analysis will still be important. But computerized tools that do not use representations, at least of the familiar sort, may now become widespread. (Not ubiquitous, though. Gould et al. (Gould, Lewis, & Becker, 1976) found that artificial notational systems were often preferred by users, even as a way to communicate with other people, for tasks like describing an assemblage of toy blocks.)

Compare "book me a room in Lund for August 21-25", and the ensuing conversational interaction, with the usual dialog process, in which one fills in forms and clicks buttons. In the usual dialog, there can certainly be a hidden dependency, or viscosity, if one can't see that information entered in one place has to match other information. For example, a very poorly designed booking system might ask for an email to be entered in two places, say in the user's account information, and then for a new booking, and be unable to determine which address to use, if these do not match.

Could something similar happen in a conversational interaction? Logically, these things could happen, but would be resolved in a natural way. That is, hidden dependencies cause problems because the system governed by artificial semantics usually can't work out what parts of an ill-formed expression to modify to make it meaningful. This is seen routinely when compilers find errors in source code. The programmer has to make the repair, and it is hard to do that if you can't see the dependency.

In a conversational interaction, the meaning of what is said is worked out as the conversation proceeds, and the system is responsible for resolving conflicts. In the booking example, a conversational system would detect the mismatch and ask which address to use.

Relatedly, viscosity happens when the programmer has to make needed changes in other parts of a

program, when a given part is changed. In a system with natural semantics more adjustments can be made without user intervention.

On the other hand, breakdowns of the kind that are common in artificial semantics, when expressions can't be interpreted, can happen in natural semantics as well. The expression "the set of all sets that do not include themselves" is an example. But in natural semantics such problems are pushed out to the edges of common communication. Life can't proceed with discourse in which breakdowns are frequent. Ideas that lack expressions that can be interpreted as needed don't get talked about much.

Natural semantics works better the more comprehensive the system's semantic grasp is. As we've seen, an LLM has semantic grasp that includes aspects of a problem domain like vandalism, as well as of computational domains like HTML. This enables it to make many choices in code, based on a single user intention, either when an initial, high level request is made ("give me a Web page for reporting vandalism", or in response to a request for modification ("make it easier to file a report").

Viscosity in an artificial semantic system like a programming language is likely still relevant, even in a world of LLMs. The more complex the constraints in a language are, that is, the dependencies among parts of a program, the harder it may be for an LLM to obtain command of the relationships between user intentions and working code, when modeling a corpus of examples. But these difficulties are no longer the direct concern of users.

7.3. Psychology of Programming, Psychology of Natural Semantics

Since, until recently, programming has been limited to artificial semantics, the psychology of programming has been the psychology of artificial semantics. Key questions have been, are some forms of artificial semantics easier to work with than others? Are there psychological interpretations of the difficulties programmers face in working with artificial semantics, and can these suggest helpful interventions? How can people be assisted in learning a system of artificial semantics?

To the extent that some programming is no longer based on artificial semantics, these questions lose importance. In the short to medium term, it appears that the natural semantics of LLMs isn't up to the job of supporting the full range of programming, or software engineering, tasks, though the limits of their eventual or even immediate applicability remain quite unclear. For now, a lot of professional programming won't be eliminated.

But much work in the psychology of programming has focused on *end user programming*, including programming in school settings for which programming knowledge itself is not the focus. As natural semantics becomes more widely useful in programming, will the psychology of natural semantics become a central part of the psychology of programming? If so, there is a simple and natural direction such a possible shift can be accommodated: we'll study what actually happens when people try to use LLMs to create programs. No doubt there will be problems.

We can already tell what some of them are. LLMs produce code that looks correct, whether it is or not, and skill is often needed to tell that the code is not actually correct. Technical approaches are emerging that seem likely to help with this. In particular, it is possible to arrange for an LLM to generate its own test cases, and to arrange to have these run, and the results interpreted and responded to by the LLM.

Beyond identifying and responding to problems, we can turn more attention to two questions that we've not been able to give much focus to, as long as we needed to deal with the complex of problems created by artificial semantics. How should one decide what program, if any, to create, in a given situation? What intellectual value does knowing how to program have in itself, rather than as a means to an end?

Acknowledgements

I thank Tamer Amin, Antranig Basman, Alan Blackwell, Luke Church, Andrea diSessa, Jonathan Grudin, Katherine Hermann, Owen Lewis, and Jason White for many valuable conversations.

8. References

- Agre, P. E. (1994). Surveillance and capture: Two models of privacy. *The information society*, 10(2), 101–127.
- Ahadi, A., Lister, R., & Teague, D. (2014). Falling behind early and staying behind when learning to program. In *Ppig* (Vol. 14).
- Anderson, J. R., & Thompson, R. (1989). Use of analogy in a production system architecture.
- Blackwell, A., & Green, T. (2003). Notational systems—the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science*. Morgan Kaufmann, 234.
- Blackwell, A. F. (1996). Metaphor or analogy: how should we see programming abstractions. In *Ppig* (p. 8).
- Brand, C. L. (2016). Programming without code: A work in-progress paper. In *Ppig*.
- Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Dabis, J., Finn, C., . . . others (2022). Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*.
- Church, L., Söderberg, E., & McCabe, A. T. (2021). Breaking down and making up-a lens for conversing with compilers. In *Ppig*.
- Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31(11), 1268–1287.
- Dershowitz, N. (1986). Programming by analogy. *Machine learning-an artificial intelligence approach*, 2, 393–422.
- DiSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and instruction*, 10(2-3), 105–225.
- diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, 29(9), 859–868.
- Garfinkel, H. (1969). *Studies in ethnomethodology*. Prentice-Hall.
- Good, J., & Brna, P. (1999). Getting a grip on the comprehension of data-flow visual programming languages. In *Ppig* (p. 7).
- Gould, J. D., Lewis, C., & Becker, C. A. (1976). *Writing and following procedural, descriptive, and restricted syntax language instructions*. IBM Thomas J. Watson Research Division.
- Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.
- Henisz-Dostert, B., & Thompson, F. (1969). A rapidly extensible language system: Rel english. In *1969 international conference on computational linguistics*.
- Heritage, J. (2013). *Garfinkel and ethnomethodology*. John Wiley & Sons.
- Kendon, A. (1994). Do gestures communicate? a review. *Research on language and social interaction*, 27(3), 175–200.
- Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., & Shah, M. (2022). Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s), 1–41.
- Lewis, C. (1988). Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science*, 12(2), 211–256.
- Lewis, C. (2019). Why can't programming be like sketching? In *Companion proceedings of the 3rd international conference on the art, science, and engineering of programming* (pp. 1–6).
- Lewis, C. (2021). *Codex as a personal assistant?* (https://docs.google.com/document/d/1b3LqmZa0dhk46W0Ry1kS7uxP_WjphCpbw-HdKgIWaXM/edit?usp=sharing)
- Lewis, C. (2022). Automatic programming and education. In *Companion proceedings of the 6th international conference on the art, science, and engineering of programming* (pp. 70–80).
- Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. In *End user development* (pp. 31–50). Springer.
- Pane, J. F., Myers, B. A., et al. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2), 237–264.

- Petre, M., & Van Der Hoek, A. (2013). *Software designers in action: A human-centric look at design work*. CRC Press.
- Roschelle, J., & Mason, J. (1995). Sketching a multidisciplinary microworld: A collaborative exploration in boxer. In *Computers and exploratory learning* (pp. 289–304). Springer.
- Rosian, V. A. (2022). Intuition-enhancing gui for visual programming. In *Ppig* (pp. 39–46).
- Simonyi, C., Christerson, M., & Clifford, S. (2006). Intentional software. In *Proceedings of the 21st annual acm sigplan conference on object-oriented programming systems, languages, and applications* (pp. 451–464).
- Van Brummelen, J., Weng, K., Lin, P., & Yeo, C. (2020). Convo: What does conversational programming need? In *2020 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 1–5).
- Webb, T., Holyoak, K. J., & Lu, H. (2023). Emergent analogical reasoning in large language models. *Nature Human Behaviour*, 1–16.