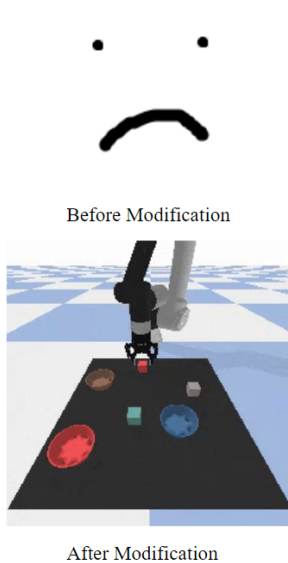


# Modified Code as Policies

David Palumbo, Joanne Wang, William Xie



**Q:** Can you move the blocks into a house?

**CaP:** outputs one of

- 1) Sure - moving the blocks into a house. (error)
- 2) No, I can only move blocks around
- 3) Code generation error

**Q:** Can you move the blocks into a house?

**MCaP:** *What shape* do you want the blocks to be in?

**Q:** A triangle.

**MCaP:** Got it - making a triangle of blocks.

## I. Abstract

Modified Code as Policies (MCaP) is a system for robot code generation policy derived from *Code as Policies: Language Model Programs for Embodied Control* [1]. In the original system, authors use large language models that are trained on code completion to produce robot policy code given natural language commands. The goal of Modified Code as Policies is to add system awareness and new low-level primitive actions so that it may handle complex, abstract, or ambiguous instructions with grace, which the original Code as Policies (CaP) will either fail to do or “hallucinate” a policy. The idea behind closed-loop human interaction is from *Inner Monologue: Embodied Reasoning through Planning with Language Models* [2]. This paper investigates the extent to which the reasoning capabilities of LLMs can be used for planning and interaction with robots. By incorporating Inner Monologue with Code as Policies, MCaP seeks out further human elaboration to remove ambiguity in itself, combined with a broader toolset, allowing it to perform better on more complex tasks.

## II. Related Work

MCaP heavily leverages large language models (LLM), primarily GPT-3, and prior work from Google Robotics [1, 2].

### Robot Policy Generation & Prompt Engineering

Prompt engineering is an emerging field of research that focuses on the development of techniques to improve the performance of natural language processing (NLP) models, such as those based on Generative Pre-trained Transformer 3 (GPT-3). GPT-3 is a recently developed

language model that is trained on a large corpus of text and is capable of generating human-like natural language responses to prompts. Prompt engineering techniques are used to refine the input to GPT-3 models, allowing them to better understand the intent of the prompt and generate more accurate and relevant responses.

Inner Monologue [2] investigates how Large Language Models (LLMs) can be used in embodied contexts to reason over sources of natural language feedback. It suggests that by leveraging environment feedback, LLMs are able to form an inner monologue that allows them to more richly process and plan in robotic control scenarios. Experiments show that closed-loop language feedback significantly improves high level instruction completion on simulated and real tabletop rearrangement tasks and long-horizon mobile manipulation tasks in a real kitchen environment. Three types of feedback are employed: Success Detection - binary classification of whether a skill has succeeded; Passive Scene Description - sources of environment grounding feedback such as object recognition; and Active Scene Description - open-ended questions answered by a person. In MCaP, we seek to incorporate Active Scene Description feedback into CaP to achieve some of the emergent capabilities from Inner Monologue: the ability to adapt to new instructions mid-task, as well as propose alternative goals when the original goal becomes infeasible.

### **Code Generation**

Codex is a code generation model for GPT-3. It is designed to generate computer code from natural language descriptions. Codex enables developers to quickly generate code for a range of tasks, such as web development, software engineering, and data analysis.

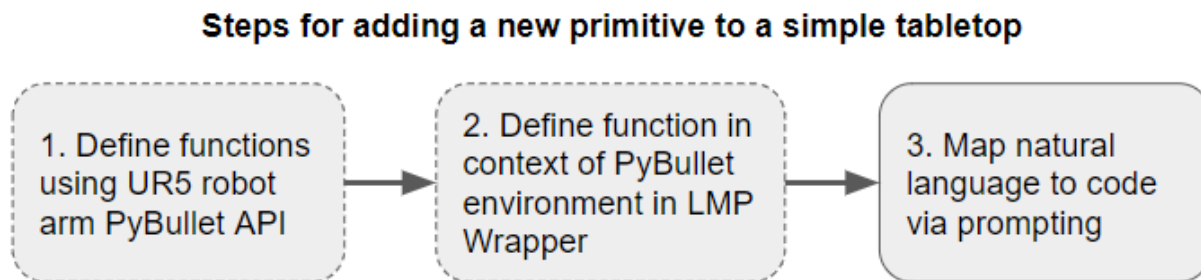
Code as Policies [1] repurposes Codex and is a system for a robot-centric formalization of language model generated programs (LMPs) that can represent reactive policies, such as impedance controllers, as well as waypoint-based policies, such as vision-based pick and place and trajectory-based control. To do this, LLMs trained on code-completion are re-purposed to write robot policy code given natural language commands. LLMs are prompted with example language commands and corresponding policy code and are able to autonomously re-compose API calls to generate new policy code. This approach is able to produce code that exhibits spatial-geometric reasoning, generalizes to new instructions, and can prescribe precise values for ambiguous descriptions. Results show that this approach outperforms existing methods on the HumanEval benchmark. However, CaP currently has a few limitations: difficulty with long horizon tasks, limited number of primitive actions, and complex (long-horizon), abstract, or ambiguous instructions. In MCaP we investigate the process of adding new primitive actions and modes of interaction to improve handling of difficult instructions.

### **III. Method**

We do this in two steps: 1. Adding new primitives “just” and “drag” to the system to enable a more robust control and 2. Allowing CaP to ask questions back to the user when facing ambiguous tasks. Our selected environment is a UR5 collaborative arm working with three blocks and three bowls on a 12” x 12” board within a PyBullet simulator. The arm has no vision

capabilities and is instead able to access ground-truth locations of objects. We generate code via OpenAI’s Codex model (code-davinci-002).

### Prompt Modification for New Primitives



**Fig. 1 Adding new primitives** is at most a three step process. When the provided API is rich enough, step 1 is not often required. When the new instruction is an issue of semantic remapping or definition, step 2 is not often required. This is, however, still a fairly manual process that scales in effort with the complexity of the environment.

The interest in the “just” command came from the system’s inability to act on objects below the top of a stack. The desired robot approach that we thought made the most sense for such a task was to extract all objects above the target, move them to a temporary location, take our action on the target, and then move the other objects back to where they were originally. We felt this is the closest to how a human would approach the task if they only had one arm, similar to the simulation system robot we were working with. Similarly with “drag,” we imagined potential use cases where the user would want to drag something along the table vs. picking it up and placing it on the table. By adding these two keywords to the system, we wanted to test the pipeline for code generation and analyze methods for making the system more robust.

For this, we took the approach of prompt engineering. Adding additional details to the original prompt passed into the system would possibly allow the system to have some background information on what was expected of itself on such a user input. See Appendix 1 & 2 for the code that was added to the prompt.

When running the model with these new prompts, we observed that the model was generating the exact code we wanted when we passed it an input such as, “Place just the brown block on the pink block” [Appendix 3] or “Drag the green block to the bottom left corner” [Appendix 4].

### Function Modifications for New Primitives

There were some issues with the included simulation environment code from the original CaP paper [1] that prevented the generated code from running. The tabletop simulation could not take into account a block’s Z-position when attempting to pick it up. Meaning that, even though it could now consider moving blocks away, it still was unable to actually pick them up individually. This required changes to the underlying PyBullet and LMP wrapper: a “get\_obj\_pos\_all” function was included to return all of the object’s coordinates, since the original one only returned x and y. Next, “pick\_place” and “put\_first\_on\_second” functions

were modified in order to utilize these values and pass them to the step function. Lastly, an optional parameter was added to the environment step function in the PickPlaceEnv for the pick\_z position, so that it could consider this information when necessary. Then the system was able to execute the above code in the way that one would expect it to.

The “drag” modifications ran into similar issues as “just.” Because in the original code base the robot’s Z-coordinate logic was pre-defined in the function, the system was unable to handle the key word “drag” and considered it equivalent to pick and place. In order to fix this, we had to modify the prompt so that when it encountered the word “drag,” it would call a new function, “drag\_to\_place,” which would then configure the parameters from the input and call “drag\_step,” which would actually execute the step with the Z-position fixed to the table.

### Question, Question, and Answer (QQA)

We modified the initial “tabletop” prompt—the prompt which detailed interaction between the instruction provider, the robot, and the tabletop simulation environment—with “awareness” of its own abilities and additional examples which allow MCaP to more gracefully handle **complex, abstract, or ambiguous** instructions. Currently, CaP will produce syntax errors or “hallucinate” when given such instructions; for example, asking it to build a house will generate a set of code with functions such as “get\_house\_points” and “make\_house,” when it has no conception of what a house is.

The “awareness” added to the tabletop prompt is imbued via a preface to the original prompt:

```
## You are a robot which receives instructions from me and turns it into code.
## I may ask you to move, place, or somehow manipulate certain objects.
## If you're unsure about any part of the instruction, ask me to clarify what I
mean.
## In the following example, I'll ask:
# Move me the most awesome block.
## Then you can ask for clarification like such:
# say(I see green, yellow, and brown blocks. Which is the most awesome?)
## Now, here are examples of interaction without much feedback.
```

The new examples added are 1-3 examples of the following instructions: confirming a navigation plan for obstacle avoidance, clarifying the meaning of an abstract adjective or noun in an instruction, or clarifying a particular bowl or block to be manipulated when given ambiguous instructions. The explicit prompt additions can be viewed in Appendix 6.

## IV. Result

### Prompt Engineering - “Just” and “Drag”

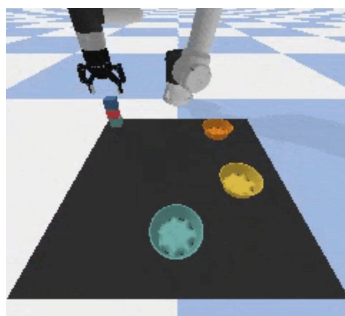
We wanted to make sure that the model was able to generalize past simply copying the given prompt using “just”, so we tested multiple other similar keywords. These included, “only”, “exclusively”, and “solely”. All of them worked as expected, generating the same code that would have been generated using “just”. This allowed us to make the assumption that the used text-davinci-002 model was adequate in understanding synonyms. This is important for a future real deployment as many users might input slightly different inputs for the same tasks.

We also wanted to test what would happen when there was more than one block stacked on top of our target, or zero on top of the target. We observed, as expected, that the same code was generated, and it was capable of moving all blocks off, or no blocks off, of our target and moving our target to the correct location without issue. Additionally, we found that the system was able to stack function calls. For instance the input “drag the cyan block to the top left corner then place the orange block on top,” would execute successfully without additional prompt modifications. This shows that the new primitives may be incorporated with other tasks for better long range performance.

We also wanted to perform exhaustion tests, to make sure that our model would perform consistently. As we expected, since the language model is independent of the environment, the arrangement of the objects in the environment had no effect on the generation of the code. We tested all keywords mentioned above, along with the double, single and zero stacks 10 times each. The code generated was the same in each case. However, we noticed that as the complexity of the input increased the system was more prone to failure. For instance the input, “Drag the green block to the bottom left corner then to the top left corner,” would fail consistently as the model cannot take in multiple parameters for “drag.” Additionally the system still has known problems such as object detection and obstacle avoidance.

## QQA

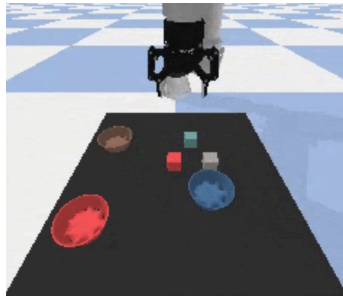
**MCaP** with the modified tabletop prompt is able to engage in a series of questions and answers. Previously, users could query the system on simple properties such as a the number or type of blocks or bowls and relative positioning, i.e. Q&A. **MCaP** now features Q&Q&A, where the system asks the user instructor for clarification or confirmation. We show results for three types of instructions: **planning** instructions, **abstract** instructions, and **ambiguous** instructions. The example MCaP results for abstract and ambiguous instructions are consistently executed with good generalizability: essentially any word that is not explicitly a block or bowl is asked for clarification, and any unspecified quantity or quality modifier is asked for clarification. However, the planning instructions can only be successfully executed consistently when the instruction matches the prompt example closely.



**Fig 2. Planning Instruction:** *The orange bowl is an obstacle. Can you move the blue block to the top while staying far away from the orange bowl?*

CaP moves the blue block to the middle of the top. MCaP moves the blue block to the shown position.

CaP	MCaP
<b>CaP:</b> Moving the blue block to the top ... code generation ...	<b>MCaP:</b> Going to move the blue block to the bottom left, then the top left. Is this a good plan? <b>Q:</b> Yes, proceed with that plan. <b>MCaP:</b> Sure - moving the blue block to the bottom left, then the top left ... code generation ...



**Fig. 3. Abstraction** instruction: *Make a fish out of the blocks*. CaP produces a syntax error. MCaP asks for clarification and executes the clarified instruction.

CaP	MCaP
<b>CaP:</b> Ok - making a fish out of the blocks ... code generation ... ... syntax error (fish polygon not found) ...	<b>MCaP:</b> I don't know what a fish is. Can you clarify? <b>Q:</b> A fish is a triangle. <b>MCaP:</b> Sure - moving the blocks into a triangle ... code generation ...

**Ambiguity** instruction: *Move the block into the bowl*. CaP arbitrarily picks a bowl and a block. MCaP asks for specification and executes the specified instruction.

CaP	MCaP
<b>CaP:</b> Moving the green block into the red bowl. ... code generation ...	<b>MCaP:</b> Which block did you mean? <b>Q:</b> The brown block <b>MCaP:</b> Got it - moving the brown block. Now, which bowl do you want to move the brown block into? <b>Q:</b> The blue bowl. <b>MCaP:</b> Moving the brown block into the blue bowl ... code generation ...

## QQA Ablation Study

The full MCaP tabletop prompt features a preface which instructs GPT-3 to ask for clarification on tasks it does not understand or may not be capable of—the “awareness”—and then adds 3 examples of planning, 3 examples of abstraction, and 2 examples of ambiguity. In our ablation study, we tested the full MCaP against a prompt with only 1 example of each type of new instruction (1-Shot MCaP) and against a prompt with no preface (Unaware MCaP).

System \ Task	Planning	Abstraction	Ambiguity
Full MCaP	Yes	Yes	Yes
1-Shot MCaP	No	Yes	Yes
Unaware MCaP	No	No	Yes

**Fig. 4 MCaP clarification success.** Success for all tasks entailed the system requesting clarification or confirmation. The specific instructions: **Planning:** “The red bowl on the top left is an obstacle. Please move the purple block left while staying far away from the obstacle.” **Abstraction:** “Please make a fish out of the blocks.” **Ambiguity:** “Please move the block into the red bowl.”

Only the full MCaP system is capable of clarifying planning instructions, and the Unaware MCaP responded to abstract instructions (i.e. requests to build a house, fish, or any unknown shape out of any unknown object) similarly to the original CaP system. This may be because the planning clarification structure is the most verbose and complex—the question and answering structure in the prompt example is the most stylistically different and most complex compared to examples from the rest of the prompt, so 1 example is not sufficient for GPT-3 to intuit that response. The Unaware MCaP prompt shows that the initial preface is key to the system responding gracefully to abstract instructions in a very generalized manner. Ambiguity is the simplest and most successful instruction because it is a matter of semantic comprehension, which LLM like GPT-3 excel at.

## V. Lessons Learned

### David

Much of the modification that was required to include new primitives was related to the fixing limitations with the simulation environment. The language model did quite well to adapt to the modifications of the prompt and was able to generalize well to similar, but not exactly identical, user inputs.

### Joanne

It was really interesting to see how large language models can be utilized in different fields outside of NLP. We found that it was an extremely tedious and strenuous process to add new primitives for more complex and robust tasks. We faced more issues with the python code than the actual utilization of the LLM. However, it was interesting to research how we can use LLMs and human reasoning as a source of input so that the model can reason through ambiguous tasks the way a human would. I think it shows really promising results and that in the future we

should look for new ways to incorporate different types of input (other than vision) to create a truly robust system.

## **William**

It is important to optimize prompt examples such that one example task might be able to generalize to tasks beyond their most obvious initial meaning (i.e. how can we add the least amount of complexity to the prompt but the greatest increase of capability for the system). This task is intentionally obfuscated by tools like GPT-3—while powerful, they limit user access to its inner workings and reduce testing to moving sliders and manual testing. Natural language is also an inherently non-optimizable, indeterminate space. Efforts to make it so are in effect efforts to make natural language rule or heuristic based language, thereby stripping it of its organic, usable nature. These two issues render a task like “prompt engineering” to currently be little more than exhaustive searches.

With that said, I have a few observations regarding prompt engineering. Architecting complex question and response structures like the planning instruction demonstrated here is not very generalizable and requires multiple examples to get right. Simple semantic remappings or clarifications ( $x$  word or phrase means  $y$  action) are trivial to accomplish. The most interesting prompts are defining simple templates or policies (for lack of a better word) in a prompt. The kind of “roleplay” established in the “awareness” preface in the QQA prompt is shown to be generalizable with almost no examples.

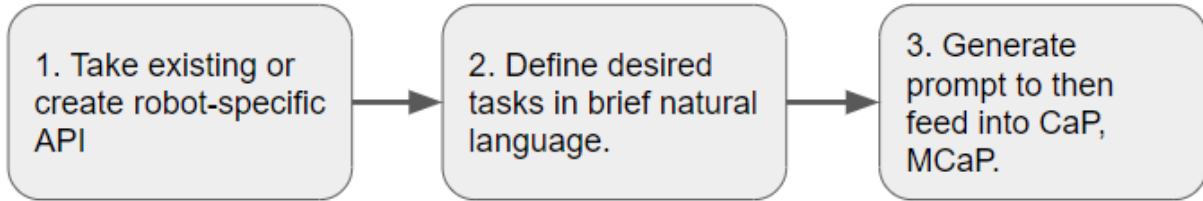
Furthermore, the prompt structure introduced with the QQA modifications approaches robot policy generation as a dynamic, collaborative system rather than a different perspective than a black-box input-output system. Because, as shown with MCaP, introducing awareness and ambiguity clarification is a low-effort task, I believe that system awareness and clarification should always be a consideration in code-generation for robot policy and collaborative robotics.

## **VI. Future Work**

Coming out of MCaP are two lines of potential inquiry. First, as noted by the original CaP authors, writing new primitives is a human resource bottleneck on extending CaP to new robot systems. One interesting and obvious extension of CaP is to shift the current paradigm. What if we ask code-generative models to generate their own primitives that they employ in policy generation, i.e. to automate Step 2 from Fig. 1? Thus a user could just pass along their existing robot control API and a set of tasks in natural language, rather than writing a LMP wrapper and large prompt, greatly reducing the initial effort to set up code policy generation.



### Proposed Automation of Constructing Primitives



Second, the QQA prompt structure is tested against a very trivial tabletop simulation environment. The systems deployed in the original CaP and Inner Monologue papers feature highly complex, real-world robot tests. The QQA prompt structure should be tested in those complex, real-world environments too for a fair comparison against those systems and in order to truly observe any emergent capabilities.

## VII. References

- [1] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng “Code as Policies: Language Model Programs for Embodied Control,” in arXiv:2209.07753, 2022
- [2] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Thompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter, “Inner monologue: Embodied reasoning through planning with language models,” in arXiv:2207.05608, 2022.

## VIII. Appendix

1. Github Repo: <https://github.com/badinkajink/modified-code-as-policies>
2. New “Just” Prompts

```
objects = ['brown bowl', 'red block', 'brown block', 'red bowl', 'pink bowl', 'pink
block']
# put the red block on the top of the pink block
say(f'Putting the red block on the pink block')
put_first_on_second('red_block', 'pink_block')
# put just the pink block on the brown block
say(f'Putting just the pink block on the brown block')
moved_objs = []
orig_pos = get_obj_pos('pink_block')
center_pos = parse_position('center of table')
for obj in objects:
    if not obj == 'pink_block':
        if on_top_of(obj, 'pink_block'):
            put_first_on_second(obj, center_pos)
            moved_objs.append(obj)
put_first_on_second('pink_block', 'brown_block')
for obj in moved_objs:
    put_first_on_second(obj, orig_pos)
```

3. New “Drag” Prompts

```
objects = ['pink block', 'green block', 'pink bowl', 'blue block', 'blue bowl',
'green bowl']
# drag the pink block to the right.
say('Sure - dragging the pink block to the right by 10 centimeters')
right_pos = parse_position('an xy point 10cm right of the pink block')
drag_to_place('pink block', right_pos)

objects = ['pink block', 'green block', 'pink bowl', 'blue block', 'blue bowl',
'green bowl']
# drag the blue block to the left
say('Sure - dragging the blue block to the left by 10 centimeters')
left_pos = parse_position('an xy point 10 cm to the left of the blue block')
drag_to_place('blue block', left_pos)
```

4. Code Generated for “Just” Input

```
user_input: "Place just the brown block on the pink block"

Running policy and recording video...
LMP tabletop_ui exec:
```

```

objects = ['pink block', 'brown block', 'green block', 'brown bowl', 'cyan bowl',
'blue bowl']
# Place just the brown block on the pink block.
say('Placing just the brown block on the pink block')
moved_objs = []
orig_pos = get_obj_pos('brown_block')
center_pos = parse_position('center of table')
for obj in objects:
    if not obj == 'brown_block':
        if on_top_of(obj, 'brown_block'):
            put_first_on_second(obj, center_pos)
            moved_objs.append(obj)
put_first_on_second('brown_block', 'pink_block')
for obj in moved_objs:
    put_first_on_second(obj, orig_pos)

robot says: Placing just the brown block on the pink block
LMP parse_position exec:

# center of table.
center_pos = denormalize_xy([0.5, 0.5])
ret_val = center_pos

```

## 5. Code Generated for “Drag” Input

```

Running policy and recording video...
LMP tabletop_ui exec:

objects = ['green block', 'orange block', 'cyan block']
# drag the cyan block to the top left corner then place the orange block on top.
say('Sure - dragging the cyan block to the top left corner then placing the orange
block on top')
top_left_corner_pos = parse_position('the top left corner')
drag_to_place('cyan block', top_left_corner_pos)
put_first_on_second('orange block', 'cyan block')

robot says: Sure - dragging the cyan block to the top left corner then placing the
orange block on top
LMP parse_position exec:

# the top left corner.
top_left_pos = denormalize_xy([0, 1])
ret_val = top_left_pos

```

## 6. 1-Shot MCoP QQA Prompt ([Link](#)). Only new sections have been included.

```

# Python 2D robot control script
## You are a robot which receives instructions from me and turns it into code.
## I may ask you to move, place, or somehow manipulate certain objects.
## If you're unsure about any part of the instruction, ask me to clarify what I
mean.
## In the following example, I'll ask:
## Move me the most awesome block.
## Then you can ask for clarification like such:
## say(I see green, yellow, and brown blocks. Which is the most awesome?)
## Now, here are examples of interaction without much feedback.

... the original prompt, lines 13 – 208 of full\_prompt ...

## Now, here are examples of interaction with feedback.
## Example 1. Other blocks or bowls can be considered as obstacles, or they can be
arbitrary.

objects = ['red block', 'green block', 'brown bowl', 'orange block', 'yellow
block', 'green bowl']
# there's now a large obstacle in the bottom. move the block near the top left to
the bottom right.

top_left_block_name = parse_obj_name('left most and top most block', f'objects =
{get_obj_names()}')
say(f'Going to move the {top_left_block_name} to the bottom left, then the bottom
right. Is this a good plan?')

# No, please plan an alternate route.

objects = ['red block', 'green block', 'brown bowl', 'orange block', 'yellow
block', 'green bowl']
say(f'Going to move the {top_left_block_name} to the top left, then the bottom
right. Is this a good plan?')

# yes, proceed with that plan.

objects = ['red block', 'green block', 'brown bowl', 'orange block', 'yellow
block', 'green bowl']
say(f'Moving {top_left_block_name} to the top left, then the bottom right.')
corner_1 = parse_position('the top left corner')
put_first_on_second(bottom_left_block_name, corner_1)
corner_2 = parse_position('the bottom right corner')
put_first_on_second(bottom_left_block_name, corner_2)

## Example 2:

objects = ['cyan block', 'white block', 'cyan bowl', 'blue block', 'blue bowl',

```

```

'white bowl']
# move the other block into cyan bowl, too.
say('The other blocks besides cyan are blue or white. Which bowl did you mean?')

objects = ['cyan block', 'white block', 'cyan bowl', 'blue block', 'blue bowl',
'white bowl']
# move the white block
say('Got it - placing the white block on the cyan bowl')
put_first_on_second('white block', 'cyan bowl')

## Example 4:

objects = ['cyan block', 'white block', 'cyan bowl', 'blue block', 'blue bowl',
'white bowl']
# move some blocks into the bowl.
say('Which blocks did you want to move?')

# I meant the cool blocks.
say('Can you clarify what the cool blocks are?')

# Cool blocks are winter-themed.
say('Got it - moving only the white block. Now, which bowl do you want to move the
white block into?')

# The bowl.
say('Can you clarify what bowl you want?')

# The cool bowl.
say('Got it - moving the white block into the white bowl.')
put_first_on_second('white block', 'white bowl')

```

7. Complexity Testing in GPT-3. In the process of system-testing CaP, we investigated LLM capabilities for handling negation. In most cases, GPT-3 (text-davinci-002) successfully interpreted simple negations (“all but” or “not”) and failed on more contrived negations (“least bottom”).

## Instructions

“undo everything it did”	Ambiguous
“move the least bottom right block to the least top right corner”	Fail (LLM)
“move the "other" blocks to the corners”	Success
“move all blocks but the green block to all corners but the corner with the green block”	Fail (LLM)

“move all blocks but the purple block to the bottom left corner”	Success
“move all blocks but the purple block to the least bottom corner”	Fail (LLM)
“move all blocks but the purple block to any corner but the bottom left corner”	Usually Success (LLM)
“move the orange block to the corner opposite of the top right corner”	Success
“spin the green block”	Fail (primitive)
“turn the brown block to the right”	Fail (primitive)
“Move the pink bowl”	Fail (LLM)