Skill-Acquisition in LLM Agents

Isai Garcia-Baza
PID: 713296738
`isaigb@live.unc.edu`

Scott Merrill
PID: 730692532
`smerrill@unc.edu`

Titus Spielvogel
PID: 730723353
`tis@unc.edu`

Abstract

We investigate how Large Language Model Agents (LLM Agents) can acquire new skills from a set of primitive actions that help to solve tasks in a specific environment. We present four changes to LearnAct (Zhao et al., 2024). Our changes mostly target the tool creation process. We evaluate our modifications on the Gripper environment and compare them to LearnAct. To investigate the effect of each proposed modification we perform an ablation study.

1 Motivation

In a blogpost (Weng) Lilian Weng summarized LLM powered agents as: an LLM + memory + planning skills + tool use. There has been a recent explosion in the research and development of LLM powered agents which might be a function of their success and open-source resources. HuggingGPT (Shen et al., 2024) augmented the action space of LLM agents to be able to call and utilize the vast set AI models on HuggingFace. Our work is motivated by the idea of expanding the action space by combining different tools. A new tool, "zoom" for example can be seen as a composition of the primitive's 'crop 'and 're-size'. Furthermore, we hope our work provides insights into how to design an LLM agents capable of combining multiple HuggingFace models to improve their performance and novelty.

2 Problem Definition

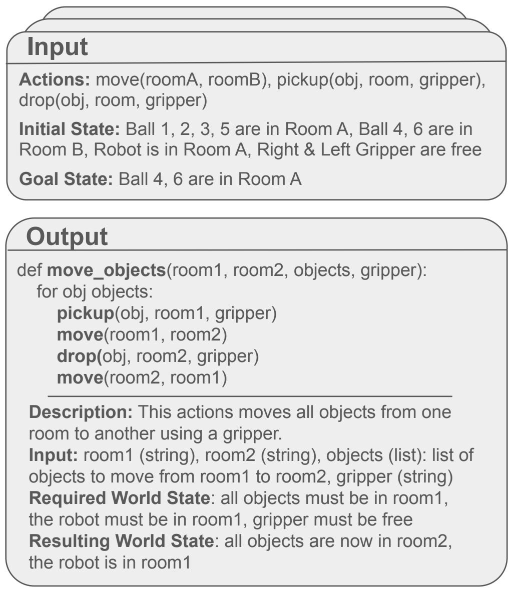
Reinforcement learning (RL) have shown impressive ability to solve Markov Decision Processes (MDPs) but often suffer in terms of sample efficiency and long training times making them less useful in practice. LLM-based agents leverage the reasoning and planning abilites of pre-trained language models to navigate these environments more efficiently. In our setting, an LLM Agent receives the current state of the environment, the goal state of the environment, and a set of predefined primitive actions all through text. The agent then must compose these primitive actions into more complex actions, allowing it to solve the task in fewer steps. At the end of the learning stage, the LLM Agent has expanded its action space with functions that help solve problems.

3 Dataset

The dataset contains 20 problems described in the Planning Domain Definition Language (PDDL) for the Gripper environment (Ma et al., 2024). The gripper environment consists of two rooms, six balls, and a robot with two grippers. The robot's goal is to move a set of balls to another room by calling three functions: move, pickup, and drop. Each problem is represented by an initial and goal state that defines the position of the balls and robot and whether the grippers are free. Three problems are used for training, and 17 problems are used for testing. Input consists of the start and goal state and the primitive actions the robot can perform. The output is a set of learned functions that compose several primitive actions. These high-level actions are specified in Python code and natural language within a docstring. An example of an Input and Output is given in Figure 1.

4 Method

Our method, shown in Figure 2, builds off the previous work done by LearnAct (Zhao et al., 2024). We borrow their 3-phase pipeline of prompting an LLM to generate tools, using those tools in the environment, and improving the tool based on their performance. We explain the create tools module in detail as this is where all our modifications to LearnAct come from. As we made only marginal modifications to the remaining modules, we only briefly describe them and instead refer the interested reader to LearnAct (Zhao et al., 2024).



Input

Actions: move(roomA, roomB), pickup(obj, room, gripper), drop(obj, room, gripper)

Initial State: Ball 1, 2, 3, 5 are in Room A, Ball 4, 6 are in Room B, Robot is in Room A, Right & Left Gripper are free

Goal State: Ball 4, 6 are in Room A

Output

```
def move_objects(room1, room2, objects, gripper):
    for obj objects:
        pickup(obj, room1, gripper)
        move(room1, room2)
        drop(obj, room2, gripper)
        move(room2, room1)
```

Description: This actions moves all objects from one room to another using a gripper.
Input: room1 (string), room2 (string), objects (list): list of objects to move from room1 to room2, gripper (string)
Required World State: all objects must be in room1, the robot must be in room1, gripper must be free
Resulting World State: all objects are now in room2, the robot is in room1

Figure 1: Example of an Input and Output

4.1 Create Tools Module

We create novel tools for our agent to use in a three-step prompting scheme utilizing two models: GPT-3.5-turbo16k (OpenAI) and codeBooga (Hugging Face, c). CodeBooga is a 32-billion parameter LLM fine-tuned on Python code. We first prompt GPT-3.5 to think of how the primitive actions can be combined to solve any arbitrary task. Specifically, we ask the model to write function descriptions for composite functions. We request that these function descriptions be complete with required inputs, input types, the necessary preconditions to use a function, and how the function changes the state of the environment. We then prompt codeBooga to write Python code based on these function descriptions. Finally, we ask GPT-3.5 if the previously generated code contains any errors and to correct any errors found. The code critique runs in a loop until the produced code has no changes or the LLM responds that the code is correct.

Our Create Tools module differentiates itself from LearnAct in four key ways. First, LearnAct prompts an LLM to think of useful functions and write Python code in a single query. We speculate asking an LLM to plan and generate code in the same step adds unnecessary difficulty to the problem. We decompose the problem into a reasoning and coding step to simplify it. In the reasoning step, an LLM must think of what functions would be useful, understand when they can be used, and how they affect the state of the environment. Second, our model differs from LearnAct in that it differentiates between reasoning and coding tasks. In all coding tasks, we query codeBooga, while LearnAct uses the same base LLM for all queries. Third, the code critic module is a novel addition to the original LearnAct model. Fourth, we made all the prompts much more directed and concise. The cumulative effect of these changes serves the code generation quality and prevents hallucinations. Each of our modifications is based on a hypothesis and two observations we made:

Hypothesis: Shorter, more directed prompts may be easier to understand. When learning tasks sequentially, Neural Networks have shown a behavior of "catastrophically forgetting" whereby earlier information is forgotten (French, 1999). While GPT-3.5 has a 16k context window, lengthy prompts may result in similar forgetting or otherwise serve to confuse an LLM. Furthermore, prompts asking an LLM to solve multiple subproblems may be fundamentally more difficult to answer.

Observation 1: LLM models fine-tuned on Python code are better at solving coding tasks, while GPT-3.5 is better at reasoning tasks. HuggingFace has several leaderboards that rank the coding ability of LLMs based on a set of unique questions. The leaderboards empirically show substantial differences between different models (Hugging Face, a).

Observation 2: Critique methods are a new and powerful approach in many LLM prompting schemes that have been shown to prevent hallucinations and improve code generation ability (Gou et al., 2023). We validate the effect of the changes proposed by our hypotheses and observations in our ablation study.

4.2 Solve Problems Module

In this module, a "user agent" is equipped with our generated tools and their corresponding descriptions. As is typically done in action prompting, we tell GPT-3.5 it operates in an action-observation-loop where it selects an action, and the environment will return an observation. We further inform the agent of its action space, initial observation, and the desired goal state. Most importantly, we provide a few shot examples of the action observation loop. We iteratively prompt the LLM to select an action until the problem is solved or a maximum number of time steps is achieved.

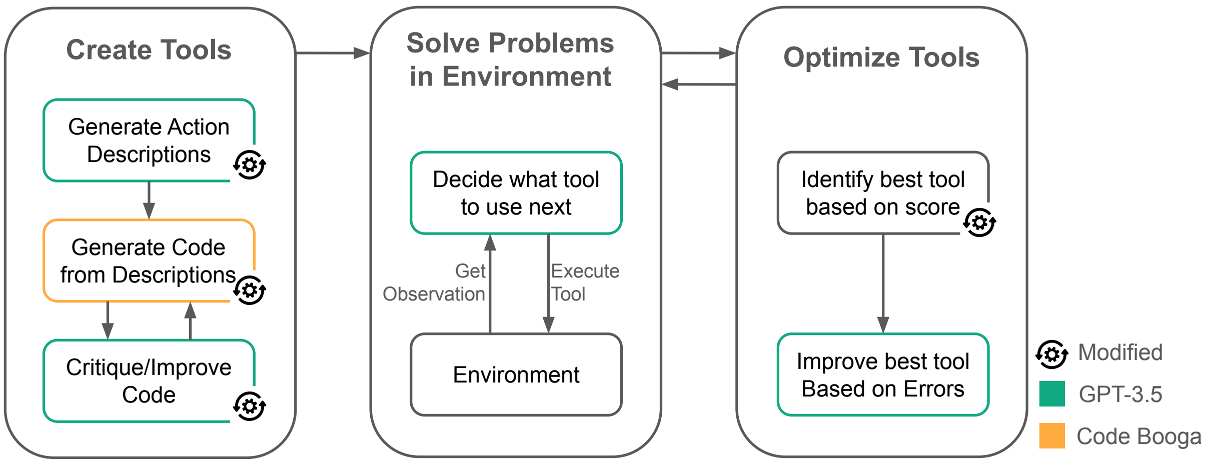

Create Tools
Generate Action Descriptions
Generate Code from Descriptions
Critique/Improve Code
Solve Problems in Environment
Decide what tool to use next
Get Observation
Execute Tool
Environment
Optimize Tools
Identify best tool based on score
Improve best tool Based on Errors
Modified
GPT-3.5
Code Booga

Figure 2: Overview of Modified LearnAct

4.3 Optimize Tools

After testing the newly generated tools in the environment, we obtained a record of useful information to further modify the tools we generated in 3.1. Specifically, we identify and trace each tool that was used and failed to determine the cause of failure, such as an invalid action, code that failed to compile, or incorrect function use. Based on this feedback, we prompt an LLM with this trace to either update the existing code to fix errors or add more detailed usage notes to prevent future misuse of that tool.

5 Experiments and Results

With our dataset size of 20, we reserved three problems (defined by a tuple of initial and goal state) for training and optimizing our toolset and 17 for testing. This is consistent with LearnAct (Zhao et al., 2024). Since we only had access to GPT-3.5 (OpenAI) we use this model for our reasoning module. We set the temperature to 0.5 and top_p=1.0. These metrics set the amount of determinism in the model. For our coding module, we used codeBooga-latest (Hugging Face, c), which scored highest on a collection of HuggingFace problems (Hugging Face, b). When using this model, we utilized the recommended settings with a temperature of 1.31, top_p of 0.14, top_k of 49, and a repetition penalty of 1.17. We ran our experiments three times and reported an average over all trials. Our results are reported in Table 1.

Task success rate measures the number of problems the agent solved within 20 time steps. A problem is solved if all objects are at the desired goal location. Average step accuracy refers to the number of valid actions an agent takes. An invalid action is one where an agent responds with an action that can't be executed by the environment. For example, an agent may try to pick up a ball that's in a different room. The average number of steps refers to the number of tool calls required to complete the task. Finally, we report the average number of tokens sent to GPT-3.5.

Table 1 shows the results of our Modified LearnAct against two baselines: the original LearnAct and Voyager (Wang et al., 2023) on the Gripper (Ma et al., 2024) environment. Our modifications resulted in a considerable increase in task success rate to 85%, which is even higher than the original LearnAct with GPT-4 as the backbone model (82.5% vs 85%). After training, the modified LearnAct Agent needed less than half the number of steps the original LearnAct agent needed, demonstrating the effectiveness of the high-level methods our modified LearnAct Agent learned. The fewer number of steps required to solve a task is due to the strong increase in average step accuracy from LearnAct (27.7%) to modified LearnAct (71.8%). Overall, the results demonstrate that our modifications increased LearnAct's overall performance. However, a more thorough analysis, including more complicated environments, such as AlfWorld (Shridhar et al., 2020) is required in the future.

Our modifications to LearnAct also resulted in fewer tokens. Specifically, our method used 2.029M tokens, which is slightly better than the 2.373M tokens LearnAct uses. The reduction in tokens is largely a result of a recursive code generation module. This module iteratively prompts

Models	Task Success Rate (GPT 3.5)	Task Success Rate (GPT 4)	Average Step Accuracy	Average Number of Steps
Baseline 1: LearnAct	45%	82.5%	27.7%	9.1
Baseline 2: Voyager	2%	76.5%	-	-
Modified LearnAct	85.0%	-	71.8%	4.3

Table 1: Test Results for Modified LearnAct and the Ablation Study on the Gripper Environment

GPT up to three times to generate code until it compiles successfully. Simplifying the code generation into two tasks allows valid code to be generated on the first pass nearly every time. Thus, breaking down the coding problem into two subproblems surprisingly reduced the total number of tokens used. Given that LLMs are priced on the basis of tokens, this improvement in token usage implies that our modifications are cheaper and more practical.

To better understand how our modifications affected the LLM agent's performance we individually tested our proposed changes on a LearnAct agent.

5.1 Ablation 1: Modified LearnAct with one step tool creation

The original LearnAct model does not decompose the task of creating tools into subtasks of creating tool descriptions and Python code. To isolate the effect of the decomposition of this task, we perform an ablation study. We use the original tool generation prompt from LearnAct but leave the error correction module and outsource code generation to codeBooga. In each trial, the task success rate was zero because codeBooga could not fully understand the problem and wrote code that could not be executed in the environment. Thus, codeBooga was unable to fully understand the task at hand. This suggests that improving fine-tuning a language model for codeing may compromise its ability to process complex natural language queries. Moreover, this highlights the fact that when using language models fine-tuned on a task, one needs to simplify the instructions and make queries as clear as possible. Our complete model provides simple queries to codeBooga by only asking it to write code based on the docstrings.

5.2 Ablation 2: Modified LearnAct without codeBooga

In our modified LearnAct we used codeBooga (Hugging Face, c), an LLM fine-tuned on Python code, for the coding of the high-level function that composes the primitive actions. We performed an ablation study, in which we analyzed the performance of or modified LearnAct without using Code Llama for the coding but GPT 3.5 instead. The results can be seen in row 5 in Table 1 (Ablation 2).

The task success rate drops from 85% to 58.8%, which indicates the superior code generation ability of codeBooga compared to GPT 3.5. The Python code for the high-level functions GPT3.5 generates is not exactly doing what the description specifies compared to codeBooga, which can also be seen in the increase in the average number of steps from 4.27 to 10.2. The step accuracy remains at around 72% (71.8% Modified Learnact vs 73.8% Modified Learnact without codeBooga).

To understand the differences of the code generated between GPT-3.5 and CodeBooga we prompt each to generate the function carry_to. We provide the exact same prompt and docustring for the function and provide the resulting code in Figure 3 of the appendix. The two generated codes are similar except but the CodeBooga function provides an extra check to ensure the robot is not in the same room as the object when the function is called. Calling move('robot_loc', 'robot_loc') would trigger an invalid action from the environment, so adding the error handling for this edge case makes the function more universally callable. Adding these error handling steps affect step accuracy in this simplified environment, however when solving more challenging problems the extra logical capacity of codebooga may be particularly valuable.

5.3 Ablation 3: LearnAct with Critique Module

The original LearnAct recursively generates new tools and directly tests them in the environment. We modified this behavior by first prompting the LLM to review its generated code and to ensure it is composed of primitive actions. This lead to a decrease of the average success of 11.76%, a step accuracy of 17.55, and 27.9 average number of steps taken out of a 30 step max. This loss in performance is in line with our hypothesis that longer prompts may confuse the LLM in problem solving tasks.

6 Conclusion

We believe our model performs better for the following reasons. Previous work has shown that LLMs are generally undertrained (Hoffmann et al., 2022). By handing off code generation to a code-focused LLM, even a smaller one, we leverage that model's higher concentration of code tokens during training. The generalist model may not have achieved this density of code tokens even if this model were many times larger. Second, our error-correcting module likely improves performance by changing the probability density function the LLM draws from. In the generation step, the LLM has likely learned to focus on the text prompt and draw from its code distribution. However, when prompted to check for mistakes, its attention is focused on the pre-generated code, and it "knows" that corrections are likely small deviations from what's given, thus a much smaller distribution to draw from.

There are several exciting areas of future work. First, our user uses action-only prompting. However, when analyzing several of our models' failed test cases, it seems that a lot can be resolved by incorporating better planning. Moreover, using a ReAct (Yao et al., 2022) loop may help the agent navigate these environments better.

References

Compuer Science Department Universtiy of North Carolina at Chapel Hill. Statement on Diversity, Equity, and Inclusion. `https://cs.unc.edu/about/dei/`. Accessed on: 04/06/2023.

Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Hugging Face. a. Big Code Models Leaderboard. `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`. Accessed on: 15/04/2023.

Hugging Face. b. CanAiCode Leaderboard. `https://huggingface.co/spaces/mike-ravkine/can-ai-code-results`. Accessed on: 02/04/2023.

Hugging Face. c. Codebooga. `https://huggingface.co/LoneStriker/CodeBooga-34B-v0.1-4.0bpw-h6-exl2`. Accessed on: 04/18/2023.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *arXiv preprint arXiv:2401.13178*.

OpenAI. GPT-3.5 Turbo. `https://platform.openai.com/docs/models/gpt-3-5-turbo`. Accessed on: 04/08/2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *CoRR*, abs/2010.03768.

The Universtiy of North Carolina at Chapel Hill. University Office for Diversity and Inclusion. `https://diversity.unc.edu/#:~:text=Our%20mission%20for%20diversity%2C%20equity,cultures%2C%20experiences%2C%20and%20perspectives.` Accessed on: 04/28/2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Lilian Weng. LLM powered autonomous agents. `https://lilianweng.github.io/posts/2023-06-23-agent/`. Accessed on: 02/04/2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Haiteng Zhao, Chang Ma, Guoyin Wang, Jing Su, Lingpeng Kong, Jingjing Xu, Zhi-Hong Deng, and Hongxia Yang. 2024. Empowering large language model agents through action learning. *arXiv preprint arXiv:2402.15809*.

A Diversity Statement

Our backgrounds are diverse in that two of us are graduate students in computer science and one in education. Two of us grew up in the US and one in Germany. We see diversity as a strength, enriching one's individual perspective. We are committed to equity, diversity, and inclusion. We welcome any feedback and look forward to opportunities to learn from one another, and improve diversity, equity, and inclusion. We fully support the statement of the computer science department on equity, diversity, and inclusion (Compuer Science Department Universtiy of North Carolina at Chapel Hill), as well as the statement of the University of North Carolina at Chapel Hill (The Universtiy of North Carolina at Chapel Hill).

B Group Members Contribution

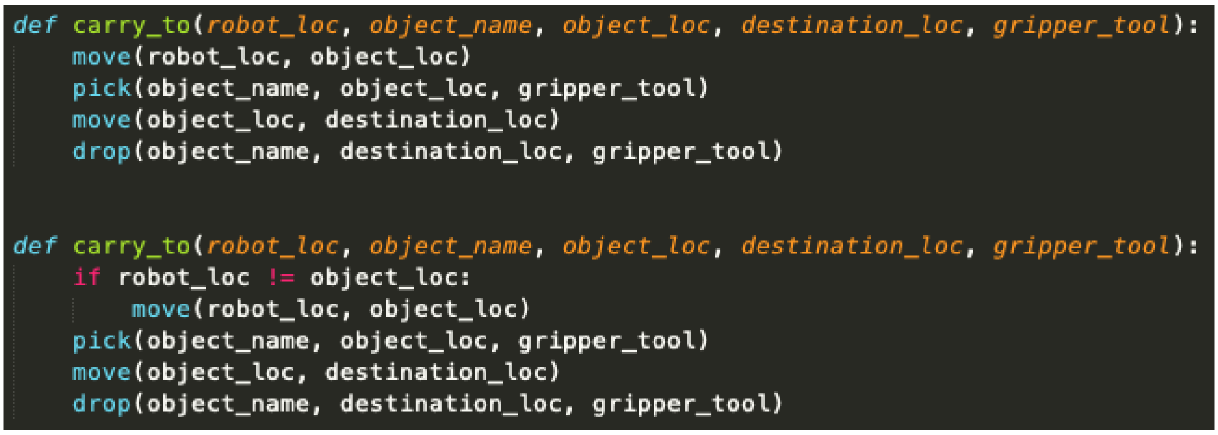
This section gives an overview of each project team member's contribution.

Scott: Helped come up with a problem direction, perform a literature review, brainstorm ideas, identify the code and helped write paper.

Isai: Proposed potential experiments and brainstorm ideas, identified text-based learning environments, helped write paper.

Titus: Helped to understand the code repo, proposed modifications, run experiments, helped writing the paper, created the figures

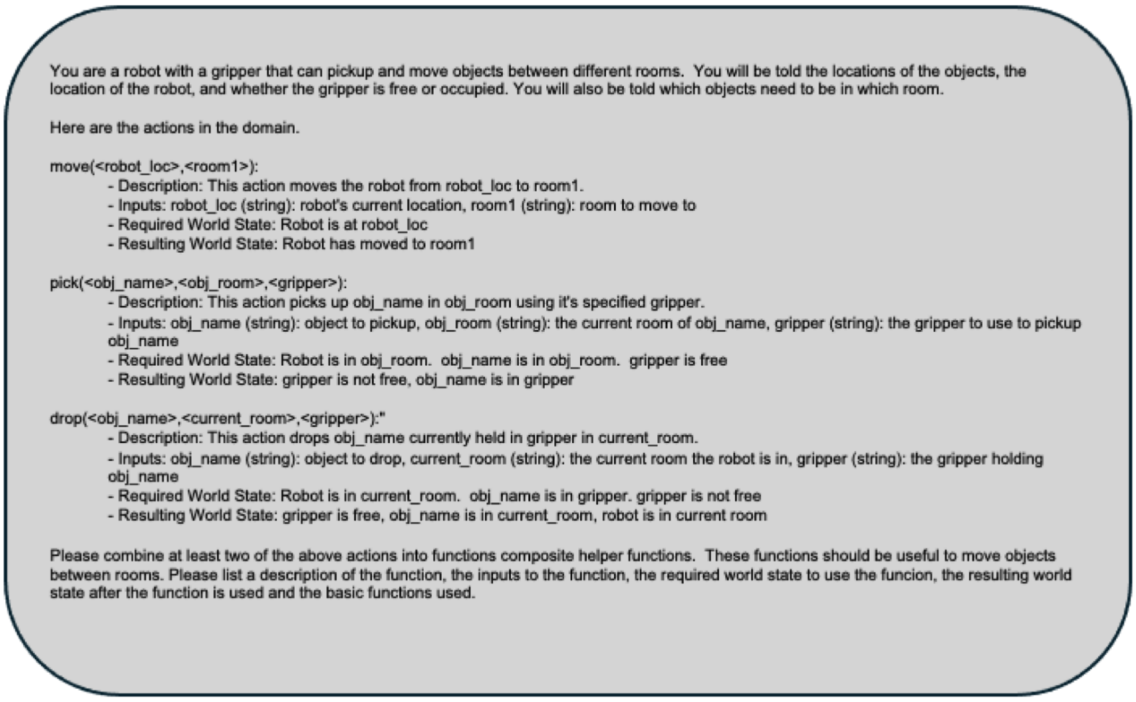
C LLM Prompts



```python
def carry_to(robot_loc, object_name, object_loc, destination_loc, gripper_tool):
    move(robot_loc, object_loc)
    pick(object_name, object_loc, gripper_tool)
    move(object_loc, destination_loc)
    drop(object_name, destination_loc, gripper_tool)


def carry_to(robot_loc, object_name, object_loc, destination_loc, gripper_tool):
    if robot_loc != object_loc:
        move(robot_loc, object_loc)
    pick(object_name, object_loc, gripper_tool)
    move(object_loc, destination_loc)
    drop(object_name, destination_loc, gripper_tool)
```

Figure 3: Comparison of CodeBooga and GPT3.5-turbo-16k Code Generation



You are a robot with a gripper that can pickup and move objects between different rooms. You will be told the locations of the objects, the location of the robot, and whether the gripper is free or occupied. You will also be told which objects need to be in which room.

Here are the actions in the domain.

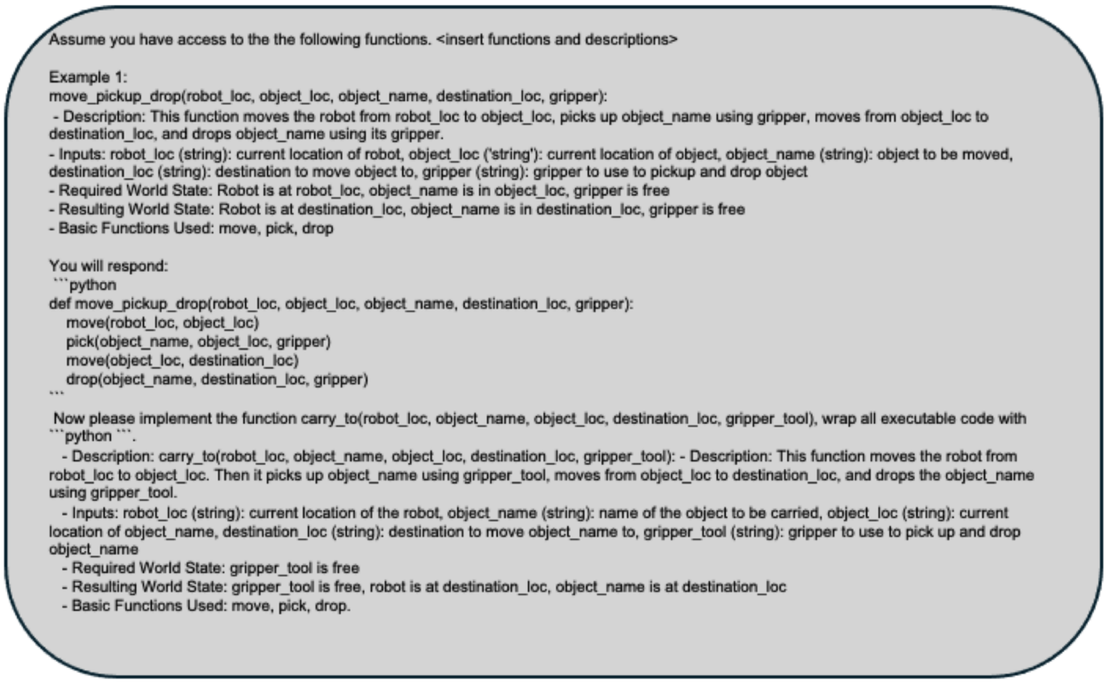
move(<robot_loc>,<room1>):
- Description: This action moves the robot from robot_loc to room1.
- Inputs: robot_loc (string): robot's current location, room1 (string): room to move to
- Required World State: Robot is at robot_loc
- Resulting World State: Robot has moved to room1

pick(<obj_name>,<obj_room>,<gripper>):
- Description: This action picks up obj_name in obj_room using it's specified gripper.
- Inputs: obj_name (string): object to pickup, obj_room (string): the current room of obj_name, gripper (string): the gripper to use to pickup obj_name
- Required World State: Robot is in obj_room. obj_name is in obj_room. gripper is free
- Resulting World State: gripper is not free, obj_name is in gripper

drop(<obj_name>,<current_room>,<gripper>):"
- Description: This action drops obj_name currently held in gripper in current_room.
- Inputs: obj_name (string): object to drop, current_room (string): the current room the robot is in, gripper (string): the gripper holding obj_name
- Required World State: Robot is in current_room. obj_name is in gripper. gripper is not free
- Resulting World State: gripper is free, obj_name is in current_room, robot is in current room

Please combine at least two of the above actions into functions composite helper functions. These functions should be useful to move objects between rooms. Please list a description of the function, the inputs to the function, the required world state to use the funcion, the resulting world state after the function is used and the basic functions used.

Figure 4: Example of Tool Description Prompt

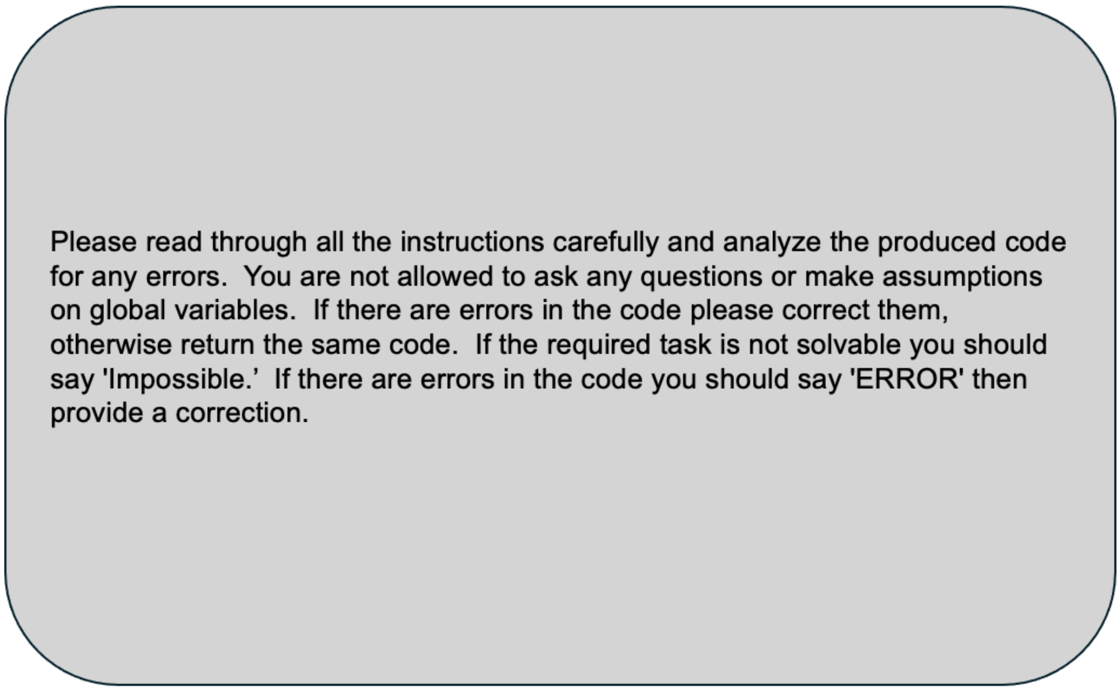


~~~
Assume you have access to the the following functions. <insert functions and descriptions>

Example 1:
move_pickup_drop(robot_loc, object_loc, object_name, destination_loc, gripper):
 - Description: This function moves the robot from robot_loc to object_loc, picks up object_name using gripper, moves from object_loc to
destination_loc, and drops object_name using its gripper.
- Inputs: robot_loc (string): current location of robot, object_loc ('string'): current location of object, object_name (string): object to be moved,
destination_loc (string): destination to move object to, gripper (string): gripper to use to pickup and drop object
- Required World State: Robot is at robot_loc, object_name is in object_loc, gripper is free
- Resulting World State: Robot is at destination_loc, object_name is in destination_loc, gripper is free
- Basic Functions Used: move, pick, drop

You will respond:
 ```python
def move_pickup_drop(robot_loc, object_loc, object_name, destination_loc, gripper):
   move(robot_loc, object_loc)
   pick(object_name, object_loc, gripper)
   move(object_loc, destination_loc)
   drop(object_name, destination_loc, gripper)
```
 Now please implement the function carry_to(robot_loc, object_name, object_loc, destination_loc, gripper_tool), wrap all executable code with
```python ```.
  - Description: carry_to(robot_loc, object_name, object_loc, destination_loc, gripper_tool): - Description: This function moves the robot from
robot_loc to object_loc. Then it picks up object_name using gripper_tool, moves from object_loc to destination_loc, and drops the object_name
using gripper_tool.
  - Inputs: robot_loc (string): current location of the robot, object_name (string): name of the object to be carried, object_loc (string): current
location of object_name, destination_loc (string): destination to move object_name to, gripper_tool (string): gripper to use to pick up and drop
object_name
  - Required World State: gripper_tool is free
  - Resulting World State: gripper_tool is free, robot is at destination_loc, object_name is at destination_loc
  - Basic Functions Used: move, pick, drop.
~~~

Figure 5: Example of Python code Prompt



Please read through all the instructions carefully and analyze the produced code for any errors. You are not allowed to ask any questions or make assumptions on global variables. If there are errors in the code please correct them, otherwise return the same code. If the required task is not solvable you should say 'Impossible.' If there are errors in the code you should say 'ERROR' then provide a correction.

Figure 6: Example of error correction prompt