INTERPRET: Interactive Predicate Learning from Language Feedback for Generalizable Task Planning

Muzhi Han[1], Yifeng Zhu[2], Song-Chun Zhu[1], Ying Nian Wu[1], Yuke Zhu[2]
[1]University of California, Los Angeles, [2]The University of Texas at Austin
https://interpret-robot.github.io

***Abstract*—Learning abstract state representations and knowledge is crucial for long-horizon robot planning. We present InterPreT, an Large Language Model (LLM)-powered framework for robots to learn symbolic predicates from language feedback of human non-experts during embodied interaction. The learned predicates provide relational abstractions of the environment state, facilitating the learning of symbolic operators that capture action preconditions and effects. By compiling the learned predicates and operators into a Planning Domain Definition Language (PDDL) domain on-the-fly, InterPreT allows effective planning toward arbitrary in-domain goals using a PDDL planner. In both simulated and real-world robot manipulation domains, we demonstrate that InterPreT reliably uncovers the key predicates and operators governing the environment dynamics. Although learned from simple training tasks, these predicates and operators exhibit strong generalization to novel tasks with significantly higher complexity. In the most challenging generalization setting, InterPreT attains success rates of 73% in simulation and 40% in the real world, substantially outperforming baseline methods.**

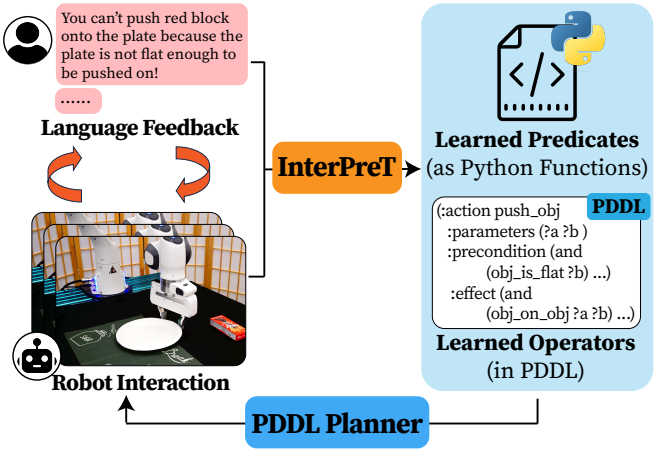


Fig. 1: **InterPreT learns predicates as Python functions and operators in PDDL from human language feedback during embodied interaction.** The learned predicates and operators can be leveraged by a PDDL planner for planning for unseen tasks involving more objects and novel goals.

I. INTRODUCTION

Effective long-horizon planning is a long-standing challenge in robotics [1, 2, 3]. Imagine a household robot that prepares a meal in your kitchen. It must be capable of generating faithful multi-step action plans to manipulate novel objects and achieve diverse task goals. Recently, Large Language Models (LLMs) have shown the ability to decompose a high-level task goal into semantically meaningful sub-tasks leveraging the vast amount of world knowledge they encode [4, 5]. They exhibit the emergent property of acquiring planning capabilities from a few in-context examples [6, 5]. Researchers have successfully applied LLM-based planners in real-world robotic tasks [7, 8, 9, 10], where they can easily incorporate various forms of feedback and produce plans in novel situations. Nevertheless, LLM-based planners still struggle to generalize strongly to long-horizon tasks, and they offer no performance guarantees [11, 12, 13].

In contrast, classical planners [14, 15] based on symbolic abstractions provide complementary strengths in generating long-horizon plans with formal guarantees. At the heart of these planners are *predicates*, which are binary-valued functions that map environment states to high-level symbolic representations, *e.g.*, a function that transforms the workspace observation into semantic relations such as `on_table(apple)`. With these symbolic predicates, we can subsequently model state transitions with symbolic *operators* [16], describing the preconditions and effects of the robot's actions on the symbolic states. The predicates and operators together form a PDDL domain [17], enabling a planning algorithm to generate plans for arbitrary in-domain tasks [18]. Despite the wide adoptions of planning algorithms in robotics [19, 20, 21], these methods usually require substantial manual effort and domain expertise to meticulously design the predicates and operators, hindering their applicability to real-world problems.

To combine the best of both worlds, there has been a growing interest in integrating learning methods with planning algorithms. Notable efforts have been made to learn symbolic representations from interaction data through unsupervised learning methods [22, 23, 24, 25, 26]. However, without explicit guidance, they struggle to uncover predicates that capture task-relevant semantic relations to facilitate planning. Meanwhile, cognitive studies [27, 28] have shown that human infants are remarkably efficient in acquiring new predicate-like relational concepts, such as spatial relations for stacking blocks, from the language feedback of caregivers during physical play. Inspired by these studies, we envision an *interactive learning* scheme that will enable a robot to rapidly learn useful abstractions for planning from online human feedback.

We hypothesize that for robots to achieve human proficiency in learning predicates for planning, they must possess an ability similar to infants to learn from the rich human

language feedback in an interactive manner. Recent work has incorporated human language feedback into learning reward functions [29] and motion policies [30]. The crux of these methods is to harness the capabilities of pretrained LLMs [31], in particular GPT-4 [32], in understanding natural language input, performing reasoning [33, 34], and generating text-based responses (computer programs [35], *etc*.). Following this line of work, we present **InterPreT** (**Inter**ative **Pre**dicate Learning for **T**ask Planning), the first framework for robots to learn planning-oriented predicates from interactive language feedback, as depicted in Fig. 1. InterPreT formalizes predicate learning as generating Python functions with GPT-4, which are iteratively refined based on human language feedback. These predicates (as Python functions) can access raw environment states with Python perception APIs and freely compose logic structures and arithmetical computations (*e.g*., with NumPy) to form complex semantics. With the learned predicates, we can easily learn symbolic operators from the robot's interaction data using a cluster-and-search algorithm [36]. The learned predicates and operators are compiled into the PDDL format on the fly to be used by a planner. LLMs' capabilities of open-world text processing and symbolic planners' performance guarantees together empower our approach to generalize strongly to arbitrary tasks in the target domains.

Specifically, we consider language feedback for learning two types of planning-oriented predicates, *i.e*., goal predicates and action precondition predicates [22]. These predicates play an essential role in indicating task progress and determining action feasibility, respectively. We design a concise and natural communication protocol to incorporate this feedback:

- **Feedback for learning goal predicates**: At the beginning of each task, the human user specifies the goal, *e.g*., "put plate on table mat". Then, it signals when the robot achieves the goal and it explains any unsatisfied conditions if the robot mistakenly declares success.
- **Feedback for learning precondition predicates**: The human user verifies the feasibility of the action the robot proposes to execute next. They explain any violated preconditions if the action is infeasible, *e.g*., "you can't pick up the plate because it is too large for the gripper to grasp", or otherwise confirm that the action is feasible, *e.g*., "you can go ahead and pick up red block".

This protocol allows InterPreT to verify and refine the learned predicates from time to time, enabling predicate learning with closed-loop feedback.

In the experiments, we evaluate InterPreT's effectiveness in a suite of simulated and real-world robot manipulation domains. These domains are designed such that their dynamics can be modeled using specific predicates and operators, which the robot must uncover. We first have InterPreT learn predicates and operators by having the robot interact with a series of simple training tasks while receiving natural language feedback from human users. We then test the learned predicates and operators on harder tasks involving more objects and novel goals. We show with qualitative and quantitative results that: (i) InterPreT learns valid predicates and operators that capture essential regularities governing each domain. (ii) The learned predicates and operators allow the robot to solve challenging unseen tasks requiring combinatorial generalization, with a 73% success rate in simulation, outperforming all baselines by a large margin. (iii) InterPreT can effectively handle real-world uncertainty and complexities, operating with considerable performance in real-world robot manipulation tasks.

II. Related Work

A. Learning Symbolic Representations for Planning

Learning symbolic abstractions of complex domains for effective planning is a long-standing pursuit in the planning community [37, 38, 39, 22, 26, 25, 3, 40]. Previous methods have focused on discovering propositional [22] or predicate state symbols [23] from embodied experience. These symbols are usually acquired by composing predefined features [37, 24, 41, 25], or learning statistical [38] or neural network [26] models with clustering [39, 22, 23] or representation learning techniques [42, 43, 26]. Such learning often relies on unsupervised objectives like minimizing state reconstruction error [42, 43, 26], prediction error [38, 42, 26], bisimulation distance [41] or planning time [25]. However, these approaches struggle to capture high-level semantic relations [43, 26] and often require manual feature engineering [24, 41, 25].

Supervised learning has also been explored to ground semantic predicates to continuous observations, *e.g*., images or continuous states [44, 45]. While large-scale annotated datasets [46] are available to learn general-purpose predicate grounding models, fine-tuning with task-specific data is still needed for learned predicates to serve reasoning and planning in specific domains [47, 48, 49]. To reduce annotation needs, prior works have employed active learning [50, 51] or novel labeling techniques [52, 53], but a minimum of 500-1000 labels [50, 51] are still required per predicate.

Our work builds on this line of research in learning symbolic abstractions from interaction data and weak supervision. We mitigate limitations of unsupervised methods by learning predicates from natural language feedback. Meanwhile, we are able to learn semantic predicates as Python functions from a few data samples, leveraging the code generation capability and world knowledge of GPT-4.

B. Large Language Models-enabled Planning and Learning

Large Language Models [31] have shown remarkable abilities in encoding vast semantic knowledge and demonstrate emergent capabilities in learning, reasoning, and planning with few-shot or even zero-shot prompting [54, 33, 5]. Pretrained LLMs have been applied as planners in text-based environments with natural language instructions and feedback [34, 5, 4, 55, 56]. For *grounded* planning in realistic robotic domains, a common approach is to utilize out-of-the-box perception models to convert raw observations into textual descriptions for LLMs to consume [57, 58, 8, 59, 60, 61], or provide perception and action APIs for LLMs to generate executable programs [10, 9]. However, these perception models struggle

to capture complex task-relevant information like semantic object relations without task-specific tuning [47, 49]. Leveraging GPT-4's power, our work effectively acquires meaningful task-relevant predicates to facilitate grounded planning.

Pretrained LLMs are also leveraged to enhance robot agent intelligence by generating formatted outputs (*e.g.*, code, formal language) and refining them based on language feedback via iterative prompting. They have been used as interfaces between natural language and robotics modalities like formal planning languages [62, 11] (*e.g.*, PDDL [17]), reward functions [29, 63] and trajectories [30]. Specifically, Voyager [64] uses GPT-4 to construct an automatic curriculum and a skill library to build lifelong learning agents, while Eureka [29] and OLAF [30] leverage GPT-4 for learning from language feedback effectively by prompting. Inspired by these works, we learn predicates from language feedback by generating and iteratively refining Python functions with GPT-4.

III. Preliminaries and Problem Setup

We consider robot task planning in a continuous state space O with language goal specifications G. Without losing generality, we assume the states are factorized with respect to a set of objects E, where such information can be obtained using mainstream perception models like object detectors. The robot is equipped with a library of primitive actions A, where each $a \in A$ is parameterized by object variables and can be grounded to certain objects to produce an executable action $\underline{a}$, *e.g.*, `Pick(cup)`. Then, a task planning problem is to find a sequence of actions $\underline{a}_{1:T}$ to reach a final state o_g that satisfies a language goal $g \in G$ from an initial state $o_0 \in O$.

Following the classical planning formulation [15, 16], we aim to learn predicates Ψ to abstract the state space O into a symbolic one S for effective and generalizable planning. A predicate $\psi :=< d_\psi, f_\psi > \in \Psi$ defines a function f_ψ that captures a symbolic relation among a list of object variables, with its semantic meaning described as d_ψ. The function $f_\psi : O \times E^k \rightarrow \{0, 1\}$ takes a continuous state $o \in O$ and a list of k objects $(e_1, e_2, ..., e_k) \in E^k$ and outputs a binary value indicating whether the relation holds or not. For example, a predicate `on(a,b)` can be applied to check whether `cup` is physically on `plate`, producing a positive literal `on(cup, plate)` or a negative literal $\neg$`on(cup, plate)`. Then the symbolic state s of a continuous state o can be obtained by collecting all positive literals at state o given predicate set Ψ and object set E, denoted $s = \texttt{Parse}(o; \Psi, E)$.

With the object-factorized symbolic state space S, we model the preconditions and effects of primitive actions with symbolic operators Ω. Each symbolic operator $\omega \in \Omega$ corresponding to a primitive action a is characterized by a precondition set CON (literals must hold before executing a), and adding and deleting effect set EFF^+ and EFF^- (literals added and removed from symbolic state s after executing a). These symbolic operators are *lifted* by design, enabling the evaluation of preconditions and effects for any executable version $\underline{a}$ obtained by applying the primitive action a to any objects. With the learned predicates Ψ, we further learn the symbolic operators Ω of all primitive actions A to achieve generalizable task planning. The learned predicates and operators can be compiled into a PDDL domain. By converting a language goal $g \in G$ into a symbolic goal s_g [11, 62], such a PDDL domain can enable effective planning using an off-the-shelf classical planner [18].

IV. Method

In this section, we present the InterPreT framework that learns predicates and operators from language feedback for planning. The overall architecture is depicted in Fig. 2. There are five essential modules that operate together to empower InterPreT: (i) ***Reasoner***, which analyzes language feedback to identify new predicates and extract task-relevant information (*e.g.*, predicate labels, action preconditions), (ii) ***Coder***, which generates Python functions to ground the new predicates, (iii) ***Corrector***, which iteratively refines existing predicate functions to align their predictions to the extracted predicate labels, (iv) ***Operator Learner***, which learns operators from interaction data based on the learned predicates, and (v) ***Goal Translator***, which translates language goal specifications into symbolic goals to enable planning. Below, we elaborate on the core GPT-4-powered modules-*Reasoner*, *Coder* and *Corrector*-that enable predicate learning, and briefly introduce the rest, which are mainly adapted from existing works.

Given language feedback l_t at time step t, our objective is to learn new predicates and refine existing predicates Ψ_{t-1}, producing an updated set of predicates Ψ_t. For simplicity of notation, we denote the textual descriptions of predicates as $\{d_\psi\}$ and the corresponding predicate functions as $\{f_\psi\}$ for any predicate set Ψ. We decompose the predicate learning process at time step t into three sequential sub-steps (see Fig. 2(a)): (i) *Reasoner* identifies new predicates with descriptions $\{d_{\psi_{new}}\}$ and extracts current state literals that provide predicate labels $\{y\}$, (ii) *Coder* generates new predicate functions $\{f_{\psi_{new}}\}$, and (iii) *Corrector* refines existing predicate functions to fix execution errors and match their predictions to $\{y\}$. Formally, we summarize this process in Eq. (1):

$$\begin{aligned}
\text{Reasoner}: \{d_{\psi_{new}}\}, \{y\} &= f_{Reason}(l_t, \{d_{\psi_{t-1}}\}), \\
\{d_{\psi_t}\} &= \{d_{\psi_{new}}\} \cup \{d_{\psi_{t-1}}\} \\
\text{Coder}: \{f_{\psi_{new}}\} &= f_{Code}(\{d_{\psi_{new}}\}, \Psi_{t-1}), \\
\{f_{\hat{\psi}_t}\} &= \{f_{\psi_{new}}\} \cup \{f_{\psi_{t-1}}\}, \\
\text{Corrector}: \{f_{\psi_t}\} &= f_{Correct}(o_t, \{y\}, \{f_{\hat{\psi}_t}\}), \\
\Psi_t &= \{< d_{\psi_t}, f_{\psi_t} >\},
\end{aligned} \tag{1}$$

where f_{Reason}, f_{Code}, $f_{Correct}$ are parameterized by GPT-4 with varying prompt templates, and the initial predicate set is empty, *i.e.*, $\Psi_0 = \varnothing$. Note that we omit some of the output terms irrelevant to predicate learning for clarity. We detail the modules below and provide the complete prompt templates in the supplementary material.

A. Reasoner

The *Reasoner* module is designed to identify essential predicates and extract task-relevant information from goal-

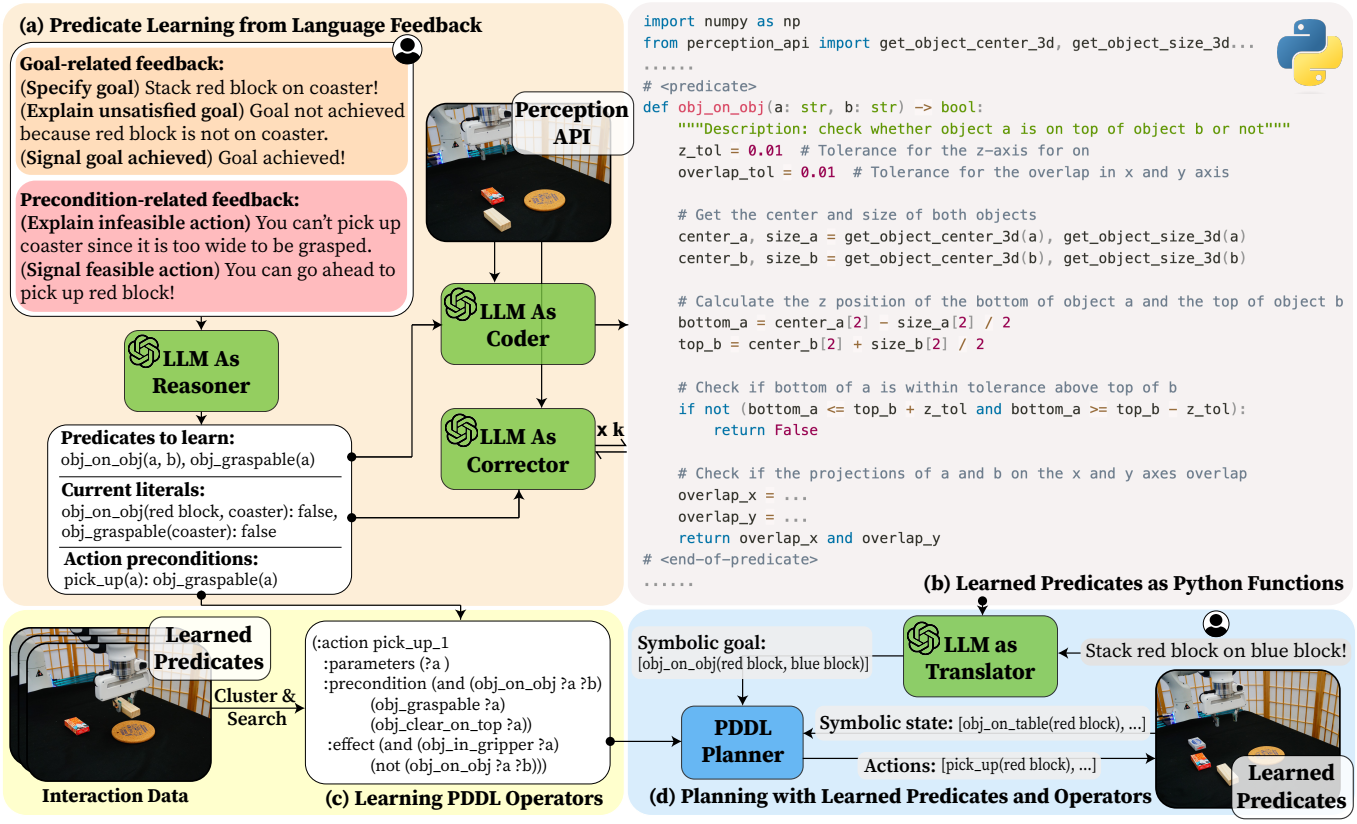


Fig. 2: **The system architecture of InterPreT.** (a) We design three GPT-4-enabled modules that operate sequentially to identify planning-oriented predicates and generate the predicate functions based on language feedback. (b) An example predicate function learned. (c) With the learned predicates, we learn PDDL operators with a cluster-then-search algorithm. (d) The learned predicates and operators enable effective task planning, after we translate language goals into symbolic goals with GPT-4.

related or precondition-related language feedback; see the top left corner of Fig. 2(a) for examples of these feedback types. We tailor *Reasoner* to each language feedback type using different prompt templates, as detailed below. We highlight the language feedback in blue, and the GPT-4 output in orange. We employ Chain-of-Thought (CoT) prompting [65] for *Reasoner* to provide the complete reasoning trace and in-context learning [33] to enable *Reasoner* to learn from a single example. These techniques are applied to all LLM promptings to facilitate robust response generation.

1) Goal-related feedback:

- **(Specify task goal)** Given a natural language goal specification, *Reasoner* identifies new goal predicates and converts the language goal into symbolic form using existing and new predicates:

> **Context:** ...{example} {objects} {existing_predicates}...
> **Goal:** Stack red block on coaster.
> **Reasoning:** The goal can be captured by a symbolic literal obj_on_obj(red block, coaster). As predicate obj_on_obj(a, b) is unknown, we need to learn it.
> **Predicates to learn:** {"obj_on_obj(a, b)": "check whether object a is on object b"}
> **Symbolic goal**: {"obj_on_obj(red block, coaster)": true}

The identified predicates to learn $\{d_{new}\}$ are to be supplied to *Coder* for predicate function generation, and the converted symbolic goal s_g is recorded for robot planning and exploration.

- **(Explain unsatisfied goal)** Given explanations for unsatisfied goal literals, *Reasoner* extracts the current symbolic literals from the language feedback, which provide predicate labels $\{y\}$. These predicate labels are crucial for *Corrector* to correct the generated goal predicates.

> **Context:** ...{example} {objects} {existing_predicates}...
> **Human explanation:** You haven't reached the goal because red block is not on coaster.
> **Current symbolic state:** {"obj_on_obj(red block, coaster)": false}

- **(Signal goal achieved)** When receiving goal-achieved signals, we simply use the extracted symbolic goal as the current literals, which provide positive labels $\{y\}$ to supervise goal predicates.

2) Precondition-related feedback:

- **(Explain infeasible action)** Given an explanation of an infeasible action, *Reasoner* identifies new precondition predicates to learn $\{d_{new}\}$, reasons about the current symbolic state $\{y\}$, and summarizes the reflected action

preconditions CON_{new}, as shown in the example below. The summarized action preconditions only take arguments that exist in those of the action, *e.g.*, for action `pick_up(a)`, we only summarize the preconditions that take no argument or object `a` as an argument. We aggregate CON_{new} into the precondition set CON_a of the corresponding action a, which is maintained for operator learning (described in Section IV-D). Additionally, $\{d_{new}\}$ is fed into *Coder* to generate precondition predicate functions, and the predicate labels $\{y\}$ are provided to *Corrector* for correction.

Context: ...{example} {objects} {existing_predicates}...
Infeasible action: pick_up(coaster)
Human explanation: You can't pick up coaster it is too large to be grasped.
Reasoning: ... The precondition of pick_up(coaster) is that it is small enough to be grasped by the gripper... We learn predicate obj_graspable(a) to check whether object a can be grasped by the gripper...
Predicates to learn: {"obj_graspable(a)": "check whether object a is small enough to be grasped by the gripper..."}
New action preconditions: {"action": "pick_up(a)", "new preconditions": {"obj_graspable(a)": true}}
Current symbolic state: {"obj_graspable(coaster)": false}

- **(Signal feasible action)** When an action is signaled as feasible, we confirm that all preconditions in CON_a are satisfied. The labels y for these precondition predicates are obtained and provided to *Corrector* to correct precondition predicate functions.

B. Coder

Once *Reasoner* identifies a set of new predicates with text descriptions $\{d_{new}\}$, the next step is to construct the corresponding predicate functions $\{f_{new}\}$ to truly learn them. Inspired by recent successes in using pretrained LLMs to generate computer programs for robotic tasks [10, 9, 64, 29], we design *Coder* to generate these predicate functions as Python code based solely on $\{d_{new}\}$, leveraging the power of GPT-4. We assume the availability of a library of perception API functions that provide access to continuous states, such as the bounding boxes and categories of detected objects. The predicate functions can then be constructed by composing these API functions with classical logic structures and arithmetical computations (*e.g.*, using NumPy), exploiting the flexibility of Python programming. Representing predicates as Python functions offers several advantages: (i) They are semantically rich and interpretable compared to neural networks [26, 42], and have better representation power and more versatile syntax than logical programs [25, 41]. (ii) They enable one-shot generation purely from the text description without labeled data, leveraging the extensive commonsense priors in GPT-4.

To facilitate the construction of predicate functions, we provide *Coder* with the following primitives: (i) perception API functions for accessing environment states, (ii) the NumPy library for arithmetic computations, and (iii) if-else and loop statements for controlling the logic structure. We also allow *Coder* to create additional utility functions that can be reused to define different predicate functions. This divide-and-conquer strategy helps mitigate the complexity of building predicate functions from scratch. In practice, we prompt GPT-4 with a code snippet demonstrating the usage of primitives by a few examples of utility functions and one example predicate function. Detailed comments are included in these examples to enable CoT prompting. Due to space limitations, we show a partial prompt with an example utility function `get_object_x_range` below, and an example predicate function in Fig. 2(b). The complete prompt for *Coder* can be found in the supplementary material.

```
import numpy as np
from perception_api import get_object_center_3d,
get_object_size_3d...

# <utility>
def get_object_x_range(a) -> np.ndarray:
    """
    Get the range of object a along x axis
    :param a: string, name of detected object
    """
    center = get_object_center_3d(a)
    extent = get_object_size_3d(a)
    return np.array([center[0] - extent[0] / 2,
    center[0] + extent[0] / 2])
# <end-of-utility>

......
```

C. Corrector

The predicate functions generated by *Coder* are pruned to two types of errors: (i) **execution errors** due to invalid operations or syntax errors in the function, and (ii) **alignment errors** that arise when GPT-4's understanding of predicates misaligns with that of the human user. Inspired by [64, 29], we introduce *Corrector*, which adopts an iterative prompting mechanism to correct the functions and rectify these errors.

When an execution error occurs while calling a predicate function, *Corrector* takes the execution trace from the code interpreter and the erroneous code as input. It then performs zero-shot reasoning on the error and generates the corrected code as output. This process is repeated until the error is resolved or the maximum number of iterations is reached. Similarly, to address alignment errors, *Corrector* is provided with the predicate labels $\{y\}$ and the current predicate functions. It then iteratively refines the functions to align their outputs with the given labels. An example prompt illustrating this process is shown below:

Context: ...{instructions} `<code>` {observation}...
Alignment error: obj_on_obj(red block, blue block) should be true, but your prediction is false.
Reasoning: This is because we assume object a should be within object b along the x-axis when we define the predicate function obj_on_obj(a, b), which is wrong; in fact, the relation is established when object a is overlapped with b, so we need to correct the function accordingly.
Corrected code: `<corrected_code>`

D. Other Components

Given the learned predicates, we implement a variant of the cluster-then-search algorithm [36] to learn operators. This algorithm effectively learns symbolic operators that best capture the action effects and preserve a minimal set of necessary preconditions from a small number of successful and failed interactions. We also incorporate the action preconditions summarized by *Reasoner* into the learned operators. To ensure learning from language feedback with no delays, we run the operator learning algorithm at each interaction step, maintaining an operator set compatible with the up-to-date predicates and interaction experience. The implementation details are included in the supplementary material.

During the training phase, InterPreT learns predicates and operators as the robot interacts with the environment to perform a series of training tasks (detailed in Section V-A3). We employ a strategy where the robot plans with a classical planner [18] based on the learned predicates and operators 50% of the time, and randomly takes a symbolically feasible action according to the recorded action preconditions otherwise. Empirically, this approach enables a balance between exploration and exploitation.

At test time, we introduce an LLM-based goal translator to convert language goals into symbolic form, following previous works [62, 11]. We refer the reader to the original papers for a detailed explanation of how the method works. In practice, we find that the GPT-4-based goal translator performs robustly when provided with a few examples.

V. Experiments

We conduct experiments to answer the following questions: (i) Can InterPreT learn meaningful task-relevant predicates and operators from language feedback? (ii) How well do the learned predicates and operators (*i.e.*, PDDL domains) generalize to tasks that involve more objects and novel goals? (iii) Can InterPreT handle perception and execution uncertainties in the real world?

A. Experimental Setup

We quantitatively and qualitatively evaluate InterPreT on a suite of robot manipulation domains in a simulated 2D kitchen environment [66] and a real-world environment. The domain design, baseline methods, and evaluation protocol are described below.

1) Domain design: We design three simulated domains based on the Kitchen2D environment [66] and two real-world domains that represent the counterpart of the simulation. Each domain is associated with a set of simple and complex tasks, and designed with a ground-truth PDDL domain file specifying the essential symbolic constraints and regularities. The five domains are each demonstrated with an example simple task in Fig. 3(a). We briefly introduce the domains below and include further details in the supplementary material.

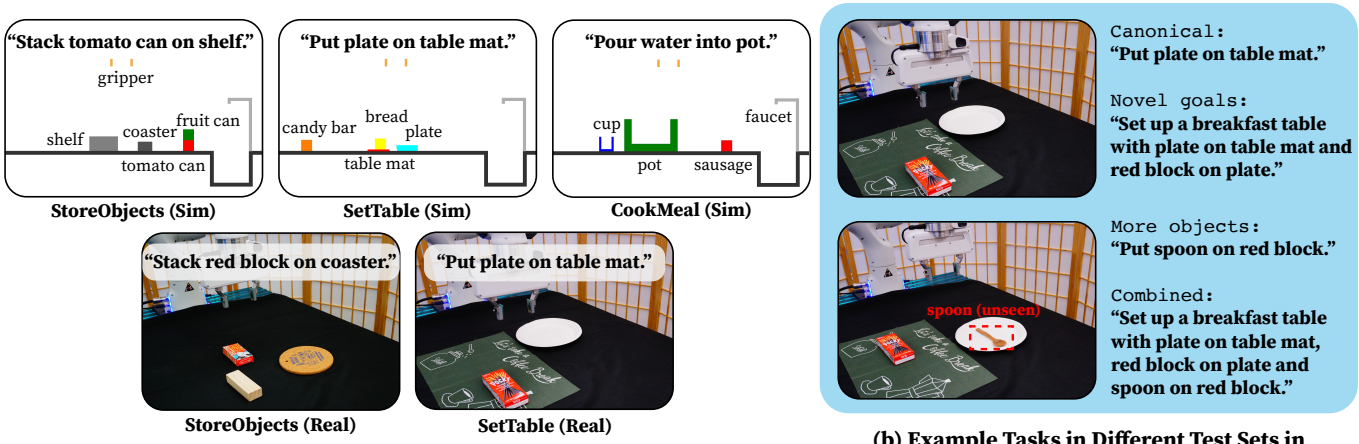
- **StoreObjects (Sim and Real)**: This domain involves storing objects on a large receptacle by picking, placing, and stacking actions. It features predicates and corresponding constraints similar to those in the BlockWorld domain [67], such as `on(a,b)`, and `on_table(a)`.
- **SetTable (Sim and Real)**: This domain involves rearranging objects to set up a breakfast table. Compared to StoreObjects, it additionally introduces a push action to move large objects (*e.g.*, plates) that cannot be grasped. It features precondition predicates such as `is_graspable(a)` and `is_flat(a)`.
- **CookMeal (Sim only)**: This domain involves putting ingredients into a pot and filling the pot/cups with water. It requires understanding the task semantics of actions, featuring predicates such as `is_container(a)`, `in(a,b)` and `has_water(a)`. It also imposes constraints such as the only way to fill a large container (*e.g.*, a pot) with water is by using a cup.

2) Baselines: As there are no prior methods that learn predicates from human language feedback for planning, we compare InterPreT with state-of-the-art LLM-based planners. (i) **Inner Monologue (IM)**[8] generates action plans based on textualized environment states using an LLM. (ii) **Code-as-Policies (CaP)**[10] employs an LLM to generate policy code that invokes perception and action APIs. We also implement variants of IM that incorporate predicates and operators learned with InterPreT. For a fair comparison, all baselines access the environment state through perception APIs and learn from in-context examples.

- **IM + Object** [8]: A naive IM variant that utilizes the textualized output of perception APIs, *e.g.*, detected objects with positions and categories, as the environment state.
- **IM + Object + Scene** [8]: An IM variant that uses environment states augmented by scene descriptions, obtained using predicates learned by InterPreT.
- **IM + Object + Scene + Precond** [8]: An IM variant that leverages the operators learned with InterPreT to check the precondition of actions proposed by IM. Infeasible actions are prompted back to the LLM for replanning.
- **CaP** [10]: A strong CaP baseline that performs precondition checks using "assertion" or if-else statements (akin to ProgPrompt[9]) and hierarchically composes policies for long-horizon planning. We have it generate predicate functions for precondition checks.

3) Evaluation protocol: We adopt a train-then-test evaluation workflow for all domains. For each domain, the robot first learns from a series of 10 simple training tasks accompanied by language feedback. For testing, we design four sets of tasks (10 tasks per set) that pose different levels of challenge to the generalizability of the methods. We present example tasks in different test sets of the real-world SetTable domain in Fig. 3(b).

- `Canonical`: Simple tasks with objects and goals seen in training but with different initial configurations.
- `More objects`: Simple tasks with seen goals but involve additional unseen objects.
- `Novel goals`: Complex tasks with seen objects but



(a) Simulated and Real-world Domains with Example Simple Tasks

(b) Example Tasks in Different Test Sets in Real-world SetTable Domain

Fig. 3: **Simulated and real-world domains used in the experiments.** We show example training tasks of all five domains in (a) and demonstrate the design of the 4 test sets in the real-world SetTable domain in (b). In `More objects` and `Combined`, an unseen object "spoon" introduces additional generalization challenges.

novel goals that compose goals seen in training tasks.

- `Combined`: Complex tasks with unseen objects and goals, combining the last two setups.

We evaluate the performance of all methods using the success rate on the 10 tasks of each test set. In simulation, we conduct systematic evaluations by running the whole training-testing pipeline 3 times with varied seeds. We directly terminate the episode upon action failure for all methods.

Domain	Goal Predicates	Precondition Predicates
StoreObjects	`obj_on_obj(a, b)`, `obj_on_table(a)`	`obj_graspable(a)`, `obj_clear(a)`, `gripper_empty()`
SetTable	`obj_on_obj(a, b)`, `obj_on_table(a)`	`obj_graspable(a)`, `obj_clear(a)`, `gripper_empty()`, `obj_is_plate(a)`, `obj_thin_enough(a)`
CookMeal	`obj_inside_obj(a, b)`, `obj_on_table(a)`, `obj_filled_with_water(a)`	`obj_graspable(a)`, `obj_clear(a)`, `gripper_empty()`, `obj_is_plate(a)`, `obj_thin_enough(a)`, `obj_large_enough(a)`, `obj_is_food(a)`, `obj_is_container(a)`

TABLE I: **Learned goal and precondition predicates in simulated domains.** We report the union of the three runs. While we learn both the positive predicate and its negated counterpart, we only show the positive ones here for clarity. We adjust some of the predicate names to unify them across domains and runs for better readability.

B. Experimental results

1) Qualitative analysis: We answer Question (i) by qualitatively analyzing the predicates and operators learned by InterPreT in the simulated domains. The full details of the learned predicate functions and operators are included in the supplementary material. Table I shows InterPreT can effectively learn language-grounded and semantically meaningful goal and precondition predicates in all three domains. We

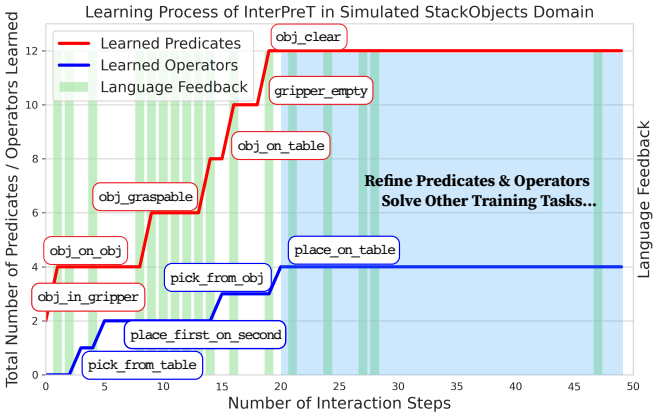


Fig. 4: **Visualization of one training run in simulated StoreObjects domain.** The total number of learned predicates increases by 2 for each labeled predicate as we also learn its negation. We provide the predicate function of `obj_in_gripper` as an in-context example, known at Step 0. We empirically label the learned operators with semantic names based on their interpreted meanings.

report the union of learned predicates over three runs; we observe that the learned predicates are generally consistent across the runs. Specifically, InterPreT successfully learns goal predicates that acquire the desired task outcomes, such as "fruit can on shelf" and "plate on table" in StoreObjects and SetTable domains and "sausage in pot" and "cup is filled" in CookMeal domain. The learned precondition predicates acutely capture the essential task constraints, such as "fruit can can only be picked up when there is nothing on its top", and "water can only be poured into a container". These well-learned predicates necessarily build the foundations for learning good operators.

We conduct a case study on one training run in the StoreObjects domain. Fig. 4 visualizes the process of learning new predicates and operators (represented as red and blue lines, respectively) while provided with intermittent language

Domain	Test Set	IM + Object [8]	IM + Object + Scene [8]	IM + Object + Scene + Precond [8]	CaP [10]	**InterPreT (Ours)**
StoreObjects	`Canonical`	0.60 ± 0.00	0.90 ± 0.00	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$	0.93 ± 0.12
	`More objects`	0.30 ± 0.00	0.83 ± 0.06	$\mathbf{1.00 \pm 0.00}$	0.83 ± 0.15	0.90 ± 0.17
	`Novel goals`	0.00 ± 0.00	0.87 ± 0.15	0.97 ± 0.06	0.53 ± 0.21	$\mathbf{1.00 \pm 0.00}$
	`Combined`	0.00 ± 0.00	0.77 ± 0.06	0.87 ± 0.15	0.03 ± 0.06	$\mathbf{1.00 \pm 0.00}$
SetTable	`Canonical`	0.80 ± 0.10	0.80 ± 0.10	$\mathbf{1.00 \pm 0.00}$	0.87 ± 0.06	$\mathbf{1.00 \pm 0.00}$
	`More objects`	0.73 ± 0.06	0.83 ± 0.12	$\mathbf{1.00 \pm 0.00}$	0.73 ± 0.15	$\mathbf{1.00 \pm 0.00}$
	`Novel goals`	0.00 ± 0.00	0.10 ± 0.10	0.53 ± 0.33	$\mathbf{0.77 \pm 0.25}$	0.53 ± 0.41
	`Combined`	0.00 ± 0.00	0.03 ± 0.05	0.20 ± 0.16	0.33 ± 0.15	$\mathbf{0.37 \pm 0.45}$
CookMeal	`Canonical`	0.90 ± 0.00	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$	0.97 ± 0.06	0.97 ± 0.06
	`More objects`	1.00 ± 0.00	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$	0.93 ± 0.06	$\mathbf{1.00 \pm 0.00}$
	`Novel goals`	0.97 ± 0.06	0.93 ± 0.06	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$
	`Combined`	0.00 ± 0.00	0.23 ± 0.15	$\mathbf{0.97 \pm 0.06}$	0.77 ± 0.12	0.83 ± 0.12
Average success rate over `Combined`		0.00	0.34	0.68	0.38	**0.73**

TABLE II: **Systematic evaluations of the methods on all test sets in simulated domains.** We highlight our method in deep gray and baselines that benefit from our learned predicates and/or operators in light grey. InterPreT achieves a 73% success rate in the most challenging `Combined` test set, outperforming all baselines by a large margin.

feedback (indicated by light green bars). Note that feedback less important for predicate learning, *i.e.*, signaling task success or feasible action, are omitted in the figure for clarity. We observe that InterPreT is able to explore effectively and acquire all predicates in 20 steps of interaction. Based on the predicates learned, InterPreT sequentially learns four operators that exhibit clear semantic meaning. Notably, it recovers two operators `pick_from_table` and `place_on_table` for the same primitive action `place_up` that is executed in different contexts. As the robot blindly explores the domain with inadequate knowledge and continuously proposes infeasible actions, dense language feedback is provided to explain precondition violations in Steps 8-20. Once InterPreT captures all action preconditions, the robot can freely navigate the environment without human intervention. Fig. 4 shows that all predicates and operators are properly initialized at Step 20 and are further corrected and refined in subsequent interactions.

2) Evaluating planning and generalization: We systematically evaluate the planning performance of all methods on the four test sets for each simulated domain. Table II presents the full results, demonstrating the strong generalizability of InterPreT when planning with a classical planner [18]. Note that several baselines utilize predicates and operators learned with InterPreT; their results are shown in light gray, while InterPreT's results are in dark gray. InterPreT achieves success rates over 90% on most test sets. On the challenging `Combined` test set, which requires strong compositional generalizability, it attains an average success rate of 73%, substantially outperforming IM variants (IM + Object, IM + Object + Scene, and IM + Object + Scene + Precond) by 73%, 39%, and 5%, respectively, and the CaP baseline by 35%.

We find the predicates and operators learned with InterPreT enable significantly improved generalization in planning, by providing meaningful relational abstractions and explicit transition modeling. The naive IM variant (IM + Object) struggles to generalize with only textualized state descriptions, solving 0% of `Combined` tasks. However, augmenting states with predicates learned by InterPreT (IM + Object + Scene) boosts the success rate on `Combined` tasks from 0% to 34%. Further ensuring action validity using learned operators (IM + Object + Scene + Precond) rivals InterPreT at 68% average success. This hybrid approach benefits from combining world knowledge in the LLM with validity guarantees from operators. However, we observe that it sometimes fails to reach the goal within the maximum number of steps due to frequent replanning. In contrast, InterPreT perform explicit PDDL planning with learned predicates and operators, and thus can generate optimal long-sequence plans with guarantee.

We demonstrate the importance of learning from language feedback by comparing InterPreT with CaP, a baseline that generates predicate functions for precondition checks and composes policies for long-horizon planning, but without leveraging language feedback. Although CaP can generate policy code with correct logic based on in-context examples, it occasionally fails to generate accurate predicate functions due to the lack of language supervision. This limitation becomes evident in CaP's poor performance on `Combined` tasks in the StoreObjects and SetTable domains, which require precise predicate understanding for successful long-horizon planning. The superior performance of InterPreT in these challenging scenarios highlights the significant benefits of incorporating natural language supervision compared to CaP.

Furthermore, we explore the transferability of learned predicates to facilitate learning and planning in new domains. We investigate the unsatisfactory performance of InterPreT in the SetTable domain, and find that simultaneously learning predicates related to pick-and-place and push actions poses a significant challenge. To address this issue, we bootstrap the learning process with predicates acquired from simpler domains. Table III demonstrates that initializing InterPreT with predicates learned in the StoreObjects domain leads to near-perfect learning in the SetTable domain, achieving 100%

	From scratch	Bootstrapped
`Canonical`	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$
`More objects`	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$
`Novel goals`	0.53 ± 0.41	$\mathbf{1.00 \pm 0.00}$
`Combined`	0.37 ± 0.45	$\mathbf{1.00 \pm 0.00}$

TABLE III: **Bootstrapping predicate learning from previously learned predicates in similar domains.** Reusing predicates learned in StoreObjects leads to near-perfect predicate learning in SetTable. The transfer of predicates is natural as all predicate functions utilize the same Perception API functions.

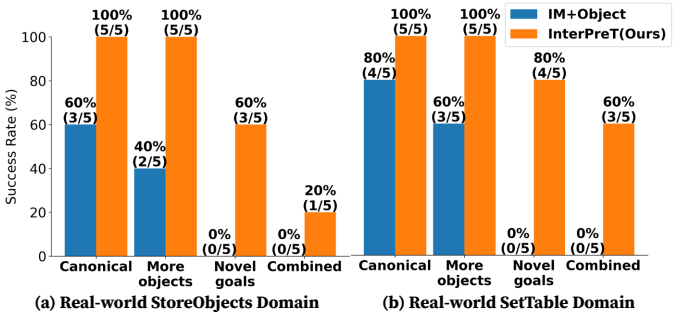


Fig. 5: **Real-robot evaluations in real-world StoreObjects and SetTable domains.** We train InterPreT once on 10 tasks and test on 5 tasks per test set. Note that the predicate learning in SetTable is bootstrapped from predicates learned in StoreObjects.

success across all test sets. This finding highlights the potential for reusing previously learned predicates to enhance learning efficiency and planning performance in complex domains.

3) Real-robot results: We evaluate InterPreT in the real-world StoreObjects and SetTable domains, compared to the vanilla IM + Object baseline [8]. We train InterPreT on 10 training tasks while a human user provides language feedback with a keyboard. We then test all methods on 5 test tasks per test set, with the success rates shown in Fig. 5. In the SetTable domain, we directly bootstrap the learning with predicates learned from StoreObjects, as the simulated results have already demonstrated the difficulty of learning from scratch in SetTable. The results indicate that InterPreT can effectively capture symbolic constraints and regularities in real-world settings where perception and execution uncertainties present. In contrast, the baseline struggles to generalize to novel task goals, highlighting the importance of the learned predicates and operators. We observe a severe performance drop for InterPreT in the StoreObjects domain under the `Combined` and `Novel goals` settings. We find that this is attributed to the increased occurrence of primitive action execution failures as the task horizon extends. Despite this, InterPreT still outperforms the baseline by a significant margin, achieving success rates of 60% and 20% in the `Novel goals` and `Combined` settings, respectively.

C. Additional Analysis and Discussions

1) Runtime Analysis: To gain insights into the computational efficiency of InterPreT, we measure its runtime in the simulated domains and provide a breakdown by stage in Table IV. Due to the variability in runtime across different iterations, we report the median, minimum, and maximum values for a comprehensive overview. The results reveal that the GPT-4-powered predicate learning and goal translation stages constitute the primary computational bottleneck. This is expected as calling the GPT-4 API involves a relatively long waiting time, which is also significantly influenced by the quality of the Internet connection. However, we anticipate that response time will cease to be a limiting factor for LLMs in the near future, given the rapid advancements in the field.

Stage		Run time / Iteration		
		Median	Min	Max
Training	Learn predicates	2.94 s	1.05 s	23.32 s
	Learn operators	32 ms	1 ms	97 ms
Testing	Translate goal	1.60 s	0.91 s	5.53 s
	Plan with PDDL	99 ms	75 ms	130 ms

TABLE IV: **Run time breakdown of InterPreT at different training and testing stages.** We show the median, minimum, and maximum values as the statistics are not normally distributed.

Domain	**#LLM Calls**	**#Transitions**	**#Feedback**
StoreObjects	22/31/23	54/75/90	17/26/18
SetTable	38/38/62	41/39/67	31/30/55
CookMeal	32/29/46	62/34/48	23/22/38

TABLE V: Number of LLM calls (**#LLM Calls**), successful state transitions collected (**#Transitions**), and language feedback provided (**#Feedback**) across the three runs in each domain.

We also present other relevant statistics in Table V, including the number of LLM calls, successful state transitions collected, and the amount of human feedback provided across three training runs for each domain. While these values exhibit considerable variation due to the inherent randomness in exploration and LLM outputs, InterPreT demonstrates the ability to recover a PDDL domain from a relatively small number of language feedback and interaction data. This highlights the sample efficiency of our approach, which is crucial for practical applications where extensive human feedback and interaction may be costly or time-consuming to obtain.

2) Robustness to varied language feedback: Natural language feedback from non-expert human users can be varied, with the same predicate being referred to in different ways. We evaluate the robustness of InterPreT to such varied feedback in the simulated StoreObjects domain by synthesizing diverse feedback templates. Using ChatGPT [68], we generate 3 alternatives for each possible feedback, which are randomly sampled during each training step. We conduct three training runs with this varied feedback and perform both qualitative and quantitative evaluations.

Table VI presents the predicates learned across the three runs, demonstrating that InterPreT robustly captures the essential goal and precondition predicates. As the goal specifications are generally consistent, InterPreT learns goal predicates with the same names in all runs. Despite the varied explanations of

	Run1	Run2	Run3
Goal Predicates	`obj_on_obj(a, b)`, `obj_on_table(a)`	`obj_on_obj(a, b)`, `obj_on_table(a)`	`obj_on_obj(a, b)`, `obj_on_table(a)`
Precondition Predicates	`obj_small_enough(a)`, `obj_clear(a)`, `gripper_empty()`	`obj_size_ok_for_gripper(a)`, `no_obj_on_top(a)`, `hand_empty()`	`obj_small_enough_for_gripper(a)`, `obj_free_of_objects(a)`, `gripper_empty()`

TABLE VI: **Learned predicates across three training runs with varied language feedback for the simulated StoreObjects domain.**

	Run1	Run2	Run3
`Canonical`	1.00	1.00	1.00
`More objects`	1.00	1.00	1.00
`Novel goals`	1.00	1.00	1.00
`Combined`	1.00	1.00	1.00

TABLE VII: **Evaluating InterPreT trained with varied language feedback for the simulated StoreObjects domain.** We report the results of all three training runs.

precondition violations, InterPreT learns precondition predicates with different names but consistent semantics. Table VII shows that the predicates and operators learned from the varied feedback yield robust planning in all test sets. These results demonstrate InterPreT's robustness to diverse language feedback, highlighting its ability to capture the underlying semantics despite variations in the feedback provided.

VI. Conclusions and Limitations

We present InterPreT, an interactive framework that enables robots to learn symbolic predicates and operators from language feedback during embodied interaction. InterPreT learns predicates as Python functions leveraging the capabilities of LLMs like GPT-4. It allows iterative correction of these learned predicate functions based on human feedback to capture the core knowledge for planning. The predicates and operators learned by InterPreT can be compiled on the fly as a PDDL domain, which enables effective task planning with a formal guarantee with a PDDL planner. Our results demonstrate that InterPreT can effectively acquire meaningful planning-oriented predicates, which allows learning operators to generalize to novel test tasks. In simulated domains, it achieves a 73% success rate on the most challenging test set that requires generalizability to more objects and novel task goals. We also show InterPreT can be applied in real-robot tasks. These findings validate our hypothesis that human-like planning proficiency requires interactive learning from rich language input, akin to infant development.

While showing promise, InterPreT has several limitations that we would like to acknowledge. First, the generalizable planning capability of InterPreT is realized by the learned symbolic operators. This introduces the assumption that the underlying domain can be well modeled at a symbolic level. This is generally not exact, as the physical world is inherently continuous. A promising future direction is to extend InterPreT into the setup of Task and Motion Planning (TAMP) [69], which considers both symbolic understanding and continuous interactions. Also, the operators learned by InterPreT are deterministic, which falls short of capturing the uncertainty in state transitions. This issue can be mitigated by learning operators with probabilistic effects [22, 26].

References

[1] Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs. In *International Conference on Robotics and Automation (ICRA)*, pages 6541–6548. IEEE, 2021.

[2] Danfei Xu, Roberto Martín-Martín, De-An Huang, Yuke Zhu, Silvio Savarese, and Li F Fei-Fei. Regression planning networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[3] Zeyu Zhang, Muzhi Han, Baoxiong Jia, Ziyuan Jiao, Yixin Zhu, Song-Chun Zhu, and Hangxin Liu. Learning a causal transition model for object cutting. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1996–2003. IEEE, 2023.

[4] Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An Huang, Ekin Akyürek, Anima Anandkumar, et al. Pre-trained language models for interactive decision-making. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:31199–31212, 2022.

[5] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning (ICML)*, pages 9118–9147. PMLR, 2022.

[6] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey for in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[7] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[8] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[9] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023.

[10] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[11] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

[12] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.

[13] Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddl planning with pretrained large language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[14] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[15] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[16] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[17] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[18] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[19] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *International Conference on Robotics and Automation (ICRA)*, pages 1470–1477. IEEE, 2011.

[20] Marc Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1930–1936, 2015.

[21] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 440–448, 2020.

[22] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Machine Learning Research (JMLR)*, 61:215–289, 2018.

[23] Steven James, Benjamin Rosman, and George Konidaris. Autonomous learning of object-centric abstractions for high-level planning. In *International Conference on Learning Representations (ICLR)*, 2021.

[24] João Loula, Tom Silver, Kelsey R Allen, and Josh Tenenbaum. Discovering a symbolic planning language from continuous experience. In *Annual Meeting of the Cognitive Science Society (CogSci)*, page 2193, 2019.

[25] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Kaelbling, and Joshua B Tenenbaum. Predicate invention for bilevel planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 37, pages 12120–12129, 2023.

[26] Alper Ahmetoglu, M Yunus Seker, Justus Piater, Erhan Oztop, and Emre Ugur. Deepsym: Deep symbol generation and rule learning for planning from unsupervised robot interaction. *Journal of Artificial Intelligence Research*, 75:709–745, 2022.

[27] Jean M Mandler. How to build a baby: Ii. conceptual primitives. *Psychological review*, 99(4):587, 1992.

[28] Tilbe Göksun, Kathy Hirsh-Pasek, and Roberta Michnick Golinkoff. Trading spaces: Carving up events for learning language. *Perspectives on Psychological Science*, 5(1):33–42, 2010.

[29] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[30] Huihan Liu, Alice Chen, Yuke Zhu, Adith Swaminathan, Andrey Kolobov, and Ching-An Cheng. Interactive robot learning from verbal correction. *arXiv preprint arXiv:2310.17555*, 2023.

[31] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[32] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[33] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:22199–22213, 2022.

[34] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2022.

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

[36] Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning symbolic operators for task and motion planning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3182–3189. IEEE, 2021.

[37] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[38] Nikolay Jetchev, Tobias Lang, and Marc Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the 2013 ICRA Workshop on Autonomous Learning*, 2013.

[39] Emre Ugur and Justus Piater. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *International Conference on Robotics and Automation (ICRA)*, pages 2627–2633. IEEE, 2015.

[40] Eric Rosen, Steven James, Sergio Orozco, Vedant Gupta, Max Merlin, Stefanie Tellex, and George Konidaris. Synthesizing navigation abstractions for planning with portable manipulation skills. In *Conference on Robot Learning (CoRL)*, pages 2278–2287. PMLR, 2023.

[41] Aidan Curtis, Tom Silver, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Kaelbling. Discovering state and action abstractions for generalized task and motion planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 36, pages 5377–5384, 2022.

[42] Elena Umili, Emanuele Antonioni, Francesco Riccio, Roberto Capobianco, Daniele Nardi, and Giuseppe De Giacomo. Learning a symbolic planning domain through the interaction with continuous environments. In *ICAPS PRL Workshop*, 2021.

[43] Masataro Asai. Unsupervised grounding of plannable first-order logic representation from images. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 583–591, 2019.

[44] Danfei Xu, Yuke Zhu, Christopher B Choy, and Li Fei-Fei. Scene graph generation by iterative message passing. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5410–5419, 2017.

[45] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*, 2019.

[46] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision (IJCV)*, 123:32–73, 2017.

[47] Xiaohan Zhang, Yan Ding, Saeid Amiri, Hao Yang, Andy Kaminski, Chad Esselink, and Shiqi Zhang. Grounding classical task planners via vision-language models. *arXiv preprint arXiv:2304.08587*, 2023.

[48] Amita Kamath, Jack Hessel, and Kai-Wei Chang. What's" up" with vision-language models? investigating their struggle with spatial reasoning. *arXiv preprint arXiv:2310.19785*, 2023.

[49] Yanjiang Guo, Yen-Jen Wang, Lihan Zha, Zheyuan Jiang, and Jianyu Chen. Doremi: Grounding language model by detecting and recovering from plan-execution misalignment. *arXiv preprint arXiv:2307.00329*, 2023.

[50] Andreea Bobu, Chris Paxton, Wei Yang, Balakumar Sundaralingam, Yu-Wei Chao, Maya Cakmak, and Dieter Fox. Learning perceptual concepts by bootstrapping from human queries. *IEEE Robotics and Automation Letters (RA-L)*, 7(4):11260–11267, 2022.

[51] Amber Li and Tom Silver. Embodied active learning of relational state abstractions for bilevel planning. In *Conference on Lifelong Learning Agents (CoLLAs)*, 2023.

[52] Toki Migimatsu and Jeannette Bohg. Grounding predicates through actions. In *International Conference on Robotics and Automation (ICRA)*, pages 3498–3504. IEEE, 2022.

[53] Jiayuan Mao, Tomás Lozano-Pérez, Josh Tenenbaum, and Leslie Kaelbling. Pdsketch: Integrated domain programming, learning, and planning. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:36972–36984, 2022.

[54] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.

[55] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[56] Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, et al. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *arXiv preprint arXiv:2311.05997*, 2023.

[57] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *International Conference on Computer Vision (ICCV)*, pages 2998–3009, 2023.

[58] Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. Embodied task planning with large language models. *arXiv preprint arXiv:2307.01848*, 2023.

[59] Yuqing Du, Ksenia Konyushkova, Misha Denil, Akhil Raju, Jessica Landon, Felix Hill, Nando de Freitas, and Serkan Cabi. Vision-language models as success detectors. *arXiv preprint arXiv:2303.07280*, 2023.

[60] Shu Wang, Muzhi Han, Ziyuan Jiao, Zeyu Zhang,

Ying Nian Wu, Song-Chun Zhu, and Hangxin Liu. Llm3:large language model-based task and motion planning with motion failure reasoning. *arXiv preprint arXiv:2403.11552*, 2024.

[61] Peiyuan Zhi, Zhiyuan Zhang, Muzhi Han, Zeyu Zhang, Zhitian Li, Ziyuan Jiao, Baoxiong Jia, and Siyuan Huang. Closed-loop open-vocabulary mobile manipulation with gpt-4v. *arXiv preprint arXiv:2404.10220*, 2024.

[62] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*, 2023.

[63] Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

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

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:24824–24837, 2022.

[66] Zi Wang, Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Active model learning and diverse action sampling for task and motion planning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 4107–4114. IEEE, 2018.

[67] Naresh Gupta, Dana S Nau, et al. Complexity results for blocks-world planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 91, pages 629–633. Citeseer, 1991.

[68] OpenAI. Chatgpt. https://chat.openai.com/.

[69] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4:265–293, 2021.

[70] Nishanth Kumar, Willie McClinton, Rohan Chitnis, Tom Silver, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Learning efficient abstract planning models that choose what to predict. In *Conference on Robot Learning (CoRL)*, pages 2070–2095. PMLR, 2023.

[71] Tianhe Ren, Shilong Liu, Ailing Zeng, Jing Lin, Kunchang Li, He Cao, Jiayu Chen, Xinyu Huang, Yukang Chen, Feng Yan, et al. Grounded sam: Assembling open-world models for diverse visual tasks. *arXiv preprint arXiv:2401.14159*, 2024.

[72] Yifeng Zhu, Abhishek Joshi, Peter Stone, and Yuke Zhu. Viola: Imitation learning for vision-based manipulation with object proposal priors. *arXiv preprint arXiv:2210.11339*, 2022.

APPENDIX A
IMPLEMENTATION DETAILS

A.I Operator Learning

With a set of known predicates, operator learning from interaction data has been studied in Task and Motion Planning (TAMP) [36, 41, 70]. In this work, we implement a vairant of the Cluster-and-search operator learning algorithm [36] to learn deterministic operators Ω from: (1) collected symbolic state transitions $\mathcal{D}_s = \{(s_{pre}, \underline{a}, s_{post})\}$ (where actions are successfully executed) and (2) action preconditions summarized from language feedback $\{\text{CON}_a\}$. Note that the symbolic transitions are pre-calculated by parsing the raw states o_{pre} and o_{post} with the learned predicates Ψ. We briefly describe how this algorithm works below.

Our operator learning algorithm aims to explain as many transitions in $\mathcal{D}_s$ with the learned operators Ω, while preserving the known action preconditions $\{\text{CON}_a\}$ in Ω. Formally, an operator ω is parameterized by a list of object variables PAR, while the lifted action a, precondition and effect sets CON, EFF^+ and EFF^- are defined with respect to PAR. A symbolic transition $(s_{pre}, \underline{a}, s_{post})$ is explained by an operator ω when there exists an assignment from PAR to specific objects in the scene, so that the grounded sets follow $\overline{\text{CON}} \in s_{pre}$, $\overline{\text{EFF}^-} = (s_{pre} - s_{post})$ and $\overline{\text{EFF}^+} = (s_{post} - s_{pre})$. We learn Ω to maximize the number of symbolic transitions explained while subject to the constraints below: (i) Each symbolic transition can be explained by at most one operator; (ii) No two operators have the same precondition sets but different effects; (iii) All operators corresponding to primitive action a must have $\{\text{CON}_a\}$ in their preconditions; (iv) All operators keep a minimal set of preconditions as possible. In practice, learning such a set of operators can be achieved in two steps following [36]: (i) initialize operators by clustering symbolic transitions by their lifted effects and preconditions, and (ii) searching for the final operators with minimal precondition set by keeping removing lifted preconditions and merging the initialized operators. We refer the readers to the original paper for more details.

A.2 Real-robot Setup

We conduct real-robot experiments on a 7-Dof Franka Panda robot arm, where a table-mounted Kinect Azure camera provides RGB-D workspace observations.

Perception APIs: We implement a set of perception APIs to retrieve the semantic and geometric information of detected objects. We detect and segment tabletop objects with Grounded-SAM [71], and use the object masks to crop the observed depth map to obtain object point clouds. Then we can estimate the geometric information, *e.g.*, 3D object bounding boxes, based on the object point clouds. To , we use higher thresholds for detection and classification, The full list of implemented perception APIs can be found in Appendix B.2.

Primitive Actions: We implement heuristic-based action primitives to physically pick, place, and push objects with the robot arm. We utilize the Deoxys [72] library for low-level control.

Handling Perception Uncertainty: To enable robust predicate learning, we also implement various strategies to handle real-world perception errors. We observe two types of errors that could happen: (i) misdetected or misclassified objects, and (ii) inaccurately estimated 3D object bounding boxes due to noisy depth maps and occlusions. Our strategies include:

- Increase detection and classification thresholds to avoid false positives. We skip the predicate learning step when a task-relevant object is not detected.
- Filter out noisy outliers for point clouds to improve 3D object bounding box accuracy.
- Prompt GPT-4 to generate predicate functions that handle uncertainties. For example, testing equality between x and b is represented as $|x - y| < tol$, where tol is a constant tolerance that is estimated and refined during iterative correction.
- Iteratively refine existing predicate functions to correct errors by prompting as the robot interacts with the environment and receives new observations and feedback.

A.3 Varied Feedback Experiment Setup

In the varied feedback experiment, we synthesize three alternatives for each language feedback template with ChatGPT [68], which are used to evaluate the robustness of InterPreT against varied language. We present the synthesized feedback templates for the StoreObjects domain as below.

- Explain infeasible action
  - Precondition 1: you can't execute `pick_up(a)` because
    - the gripper is already occupied
    - the gripper already held an object
    - the gripper has an object in hand and can't pick up more objects
  - Precondition 2: you can't execute `pick_up(a)` because
    - there is something on a

    - object a has another object on its top
    - there is another object on object a so you can't pick it up
  - Precondition 3: you can't execute `pick_up(a)` because
    - a can not be grasped by the gripper as it is too wide
    - object a is too large for the gripper to pick up
    - you can't pick up object a because it is too large
  - Precondition 4: you can't execute `place_on_table(a)` because
    - object a is not held by the gripper
    - object the gripper does not hold object a
    - object a is not in the gripper
  - Precondition 5: you can't execute `place_first_on_second(a,b)` because
    - object a is not held by the gripper
    - object the gripper does not hold object a
    - object a is not in the gripper
  - Precondition 6: you can't execute `place_first_on_second(a,b)` because
    - there is something on b
    - object b has another object on its top
    - there is another object on object b so you can't pick it up
- Explain unsatisfied goal conditions
  - You haven't achieved the goal because
    - object a is not yet on b
    - object a has not been put on object b
    - you haven't put object a on object b
  - You haven't achieved the goal because
    - object a is not on table
    - object a is not yet on table
    - you haven't put object a on table

APPENDIX B
FULL PROMPT TEMPLATES

B.1 Reasoner

We present the prompt templates of *Reasoner* for different types of language feedback below.

1) Specify goal:

```
You are a helpful assistant that converts natural language goals into symbolic goals, by
proposing necessary predicates to learn. A predicate is a function that takes objects as
arguments and outputs True or False. Please reason about predicates that directly reflect the
goals, and only include these predicates in the symbolic goal. Please try to reuse available
predicates to avoid redundant predicates. When give the symbolic goal, only include the
symbolic literals that are directly mentioned in the goal text. The literals only take
available objects as arguments (not "table", "any", or other variables). Do not infer or
guess on what other literals should be in the goal. If you can't come up with goal literals
given the above constraints, think about inventing new predicates. Other known environment
entities include "table" and "gripper", so you can invent predicates such as "in_gripper",
"under_table", etc.
Note that the predicates in the given example are unknown to you.

Example:
Available objects: ['plate']
Available predicates: {'obj_in_sink(a)': 'check whether object a is in sink or not'}
Goal: Wash the plate and move it out of the sink.
Output:
{
    "Reasoning": "The goal directly captures two symbolic literals, obj_washed(plate) and
    obj_in_sink(plate). As predicate obj_in_sink(a) is available, we only need to invent
    predicate obj_washed(a).",
```

```
    "Invented predicates": {
        "obj_washed(a)": "check whether object a is washed or not"
    },
    "Symbolic goal": {
        "obj_washed(plate)": true,
        "obj_in_sink(plate)": false
    }
}

{domain_desc}
Available objects: {entities}
Available predicates: {predicates}
Goal: {goal_spec}
Please give the output following the format in the examples, and make sure to include all
fields of the json in your outputs.
Output (please don't output ```json):
```

2) Explain unsatisfied goal:

```
You are a helpful assistant that translates human explanations of the current state into
symbolic literals, using the available predicates. The human explanations are about why the
goal is not achieved in the current state.

Example:
Available objects: ['cup', 'block', 'plate']
Available predicates: {'holding(x)': 'check whether the gripper is holding x or not',
'in_sink(x)': 'check whether x is in sink or not'}
Human explanation: The cup is not in the sink, and it is held by the gripper.
Output:
{
    "Current state": {
        "holding(cup)": true,
        "in_sink(cup)": false
    }
}

{domain_desc}
Available objects: {entities}
Available predicates: {predicates}
Human explanation: {human_explain}
Output (please don't output ```json):
```

3) Explain infeasible action:

```
You are a helpful assistant that invents predicates to describe action preconditions, based
on human explanations of why an action is infeasible. A predicate is a function that takes
objects as arguments and outputs True or False. A literal (or an atom) grounds a predicate on
available objects (but not "table", "any", or other variables).
Please generate output step-by-step:
1. Reasoning: Based on the human explanation, reason about the current state and the
preconditions of executing the action. Then invent predicates that directly represent the
action preconditions. The predicates can only take in object variables as inputs, but not
environment-related entities such as "gripper", "robot", or vague entities such as "any
object". Predicates can have empty arguments. Please try to reuse available predicates to
avoid inventing redundant predicates.
2. Invented predicates: Based on the reasoning in Step 1, list the invented predicates and
their explanations. Make sure to include as detailed explanations as possible according to
the human explanation. The predicates only take object variables as arguments (not numerical
variables).
3. New action preconditions: Based on the reasoning in Step 1, give the new preconditions of
the action that take lifted variables such as "a", "b" as arguments. Please do not include
other preconditions that are not mentioned in the human explanation.
4. Current state literals: Based on the explanation and reasoning in Step 1, give the curent
state literals mentioned in the human explanation, using the invented and existing
predicates. Please do not include other literals that are not mentioned in the human
explanation. If you are not sure about whether to include a literal, don't include it.
```

```
Note that the predicates in the given example are unknown to you.

Example:
Available objects: ['cup', 'plate']
Available predicates: {}
Infeasible action: wash(plate)
Human explanation: You can't wash plate because the robot is far away from the plate, the
robot needs to be within 1m from the plate to wash it; also, the robot has something else in
hand.
Output:
{
    "1. Reasoning": "Based on human explanation, we know the robot is farther than 1m from
    the plate, and it has something else in hand, so it can't to wash the plate. The
    precondition of action wash(plate) is that the robot is close enough (smaller than 1m) to
    the plate, and its hand is empty. Given the available predicates, none of them can
    directly represent the precondition; so we invent predicate obj_close_enough(a) to check
    whether object a is close enough, and predicate hand_empty() to check whether robot has
    its hand free. Then the current literal follows obj_close_enough(plate) is false and
    hand_empty() is false, and the action precondition is obj_close_enough(plate) is true and
    hand_empty() is true."
    "2. Invented predicates": {
        "obj_close_enough(a)": "check whether object a is close enough to the robot, the
        predicate holds when the distance between robot and object is smaller that 1m",
        "hand_empty()": "check whether the robot has its hand free, the predicate holds when
        the robot is not holding anything"
    },
    "3. New action preconditions": {
        "action": "wash(a)",
        "new preconditions": {
            "obj_close_enough(a)": true,
            "hand_empty()": true
        }
    },
    "4. Current state literals": {
        "obj_close_enough(plate)": false,
        "hand_empty()": false
    }
}

{domain_desc}
Available objects: {entities}
Available predicates: {predicates}
Infeasible action: {failed_action}
Human explanation: {failure_explain}
Please give the output following the format in the examples, and make sure to include all
fields of the json in your outputs.
Output (please don't output ```json):
```

B.2 Coder

1) Main template:

```
You are a helpful assistant that writes python functions to ground the given predicates. A
set of perception API functions are available to provide the basic perception information.
You can write extra utility functions that can be used in your predicate functions. You can
also introduce new dependent predicates when necessary, but please keep the set of predicates
as compact as possible. Note that some predicates may depend on each other; so you can reuse
predicate functions in other predicates to avoid writing duplicate code.
{domain_desc} Please find the available API functions, and examples of utility and predicate
functions below.

{known_functions}

The observation is: {observation}
```

```
Now you are asked to ground the predicates below: {new_predicates}
Please put "# <predicate>" and "# <end-of-predicate>", "# <utility>" and "# <end-of-utility>"
at the beginning and end of each predicate function and utility function. Remember to include
description surrounded with "<<", ">>" for predicates.
```

2) Code example for simulated domains:

```python
import numpy as np
from typing import *
from predicate_learning.predicate_gym.perception_api import (
    get_detected_object_list,
    get_object_xy_position,
    get_object_xy_size,
    get_object_category,
    get_object_water_amount,
    get_gripper_position,
    get_gripper_open_width,
    get_in_gripper_mass,
    get_gripper_y_size,
    get_gripper_max_open_width,
    get_table_x_range,
    get_table_y_height,
)

eps = 0.2


# Utility functions:
# <utility>
def get_object_xy_bbox(a) -> np.ndarray:
    """
    Get the xyxy bounding box of object a
    :param a: string, name of detected object
    :return: np.ndarray, [x1, y1, x2, y2], where x1 is left, x2 is right, y1 is bottom, y2 is top
    """
    object_a_position = get_object_xy_position(a)
    object_a_size = get_object_xy_size(a)
    return [
        object_a_position[0] - object_a_size[0] / 2,
        object_a_position[1] - object_a_size[1] / 2,
        object_a_position[0] + object_a_size[0] / 2,
        object_a_position[1] + object_a_size[1] / 2,
    ]
# <end-of-utility>


# <utility>
def get_in_gripper_xy_bbox() -> np.ndarray:
    """
    Get the xyxy bounding box of space within the gripper
    :param a: string, name of detected object
    :return: np.ndarray, [x1, y1, x2, y2], where x1 is left, x2 is right, y1 is bottom, y2 is top
    """
    gripper_position = get_gripper_position()
    gripper_open_width = get_gripper_open_width()
    gripper_y_size = get_gripper_y_size()
    return [
        gripper_position[0] - gripper_open_width / 2,
        gripper_position[1] - gripper_y_size / 2,
        gripper_position[0] + gripper_open_width / 2,
        gripper_position[1] + gripper_y_size / 2,
    ]
# <end-of-utility>


# Predicates:
# <predicate>
def obj_in_gripper(a) -> bool:
    """
    Description: <<whether object a is held by the gripper>>
    The predicate holds True when the mass in gripper is non-zero, object a is aligned with the opened
    gripper along x axis, and overlaps with the gripper along y axis
```

```
    :param a: string, name of detected object
    :return: bool
    """
    # get in gripper mass
    in_gripper_mass = get_in_gripper_mass()
    # get in_gripper xyxy bbox
    in_gripper_xyxy_bbox = get_in_gripper_xy_bbox()

    # get bbox_xyxy of object a
    object_a_xyxy_bbox = get_object_xy_bbox(a)

    # check whether the mass in gripper is non-zero
    # in_gripper_mass > eps
    if in_gripper_mass > eps:
        # check whether the object a is aligned with the gripper along x axis
        # abs(a.x1 - gripper.x1) < eps and abs(a.x2 - gripper.x2) < eps
        if (
            np.abs(object_a_xyxy_bbox[0] - in_gripper_xyxy_bbox[0]) < eps
            and np.abs(object_a_xyxy_bbox[2] - in_gripper_xyxy_bbox[2]) < eps
        ):
            # check whether the object a overlaps with the gripper along y axis
            # a.y1 < gripper.y2 + eps and a.y2 > gripper.y1 - eps
            if (
                object_a_xyxy_bbox[1] < in_gripper_xyxy_bbox[3] + eps
                and object_a_xyxy_bbox[3] > in_gripper_xyxy_bbox[1] - eps
            ):
                return True
            else:
                return False
        else:
            return False
    else:
        return False
# <end-of-predicate>
```

3) Code example for real-world domains:

```
import numpy as np
from typing import *
from real_robot.perception_api import (
    get_detected_object_list,
    get_object_category,
    get_object_center_3d,
    get_object_size_3d,
    get_gripper_position_3d,
    get_gripper_max_open_width,
    get_gripper_open_width,
    get_table_height,
)

# All numbers are in meters. Please reason about the proper tolerance value for each predicate to
incoporate real-world perception uncertainty. You may design different tolerance for different variables,
like along different axis.
# Usually, when objects are close to each other (e.g., stacked together), the segmented objects may
include part of each other and their bounding boxes may overlap. You need to handle this using the
tolerance properly.

# Utility functions:
# <utility>
def get_object_x_range(a) -> np.ndarray:
    """
    Get the range of object a along x axis
    :param a: string, name of detected object
    """
    center = get_object_center_3d(a)
    extent = get_object_size_3d(a)
    return np.array([center[0] - extent[0] / 2, center[0] + extent[0] / 2])
# <end-of-utility>

# Predicates:
# <predicate>
def obj_in_gripper(a) -> bool:
    """
```

```
    Description: <<whether object a is held by the gripper>>
    The predicate holds True when gripper is half open and object a is close to the gripper.
    :param a: string, name of detected object
    """
    gripper_open_tol = 0.01
    z_tol = 0.01
    xy_tol = 0.01

    # check whether gripper is half-open
    # gripper_width < max_gripper_width + gripper_open_tol
    if get_gripper_open_width() < get_gripper_max_open_width() - gripper_open_tol:
        gripper_position = get_gripper_position_3d()
        object_center = get_object_center_3d(a)
        object_extent = get_object_size_3d(a)
        # check whether the distance between object a and gripper along z is within z_tol
        # abs(gripper_z - object_center_z) < z_tol
        if np.abs(gripper_position[2] - object_center[2]) < z_tol:
            # check whether the distance between object a and gripper along x and y is within half the
            extent of a
            # abs(gripper_xy - object_center_xy) < object_extent_xy / 2 + xy_tol
            if np.all(np.abs(gripper_position[:2] - object_center[:2]) < object_extent[:2] / 2 + xy_tol):
                return True
            else:
                False
        else:
            False
    else:
        return False
# <end-of-predicate>
```

B.3 Corrector

1) Execution error:

```
You are a helpful assistant that modifies the predicate grounding functions based on the
execution error.

{domain_desc}

{known_functions}

The observation is: {observation}

The execution error is: {error}

The error trace is: {trace}

Please answer the two questions below:
1. What is your reasoning for the execution error? How would you modify the predicate
grounding functions to fix the error?
2. Based on your reasoning, please return the modified functions without further
explanations. Please put "# <predicate>" and "# <end-of-predicate>", "# <utility>" and "#
<end-of-utility>" at the beginning and end of each predicate function and utility function,
following the format of the original functions. Remember to include description surrounded
with "<<", ">>" for predicates.
```

2) Alignment error:

```
You are a helpful assistant that modifies the predicate functions based on correction from
human. You can introduce new utility functions and predicates in your modification when
necessary. Note that the inconsistency might not due to the direct implementation of the
predicate, but other predicate functions or utility functions it depends on. If this is the
case, you need to modify its dependent predicates andutility functions.
{domain_desc} The current predicate functions are below.

{known_functions}

The observation is: {observation}
```

```
The human correction is: {correction}

Please answer the two questions below:
1. What is your reasoning for the human corection? Which part of the predicate functions is
wrong? How would you modify the predicate grounding functions to reflect the correction?
2. Based on your reasoning, please return the modified functions without further
explanations. Please put "# <predicate>" and "# <end-of-predicate>", "# <utility>" and "#
<end-of-utility>" at the beginning and end of each predicate function and utility function,
following the format of the original functions. Remember to include description surrounded
with "<<", ">>" for predicates.
```

B.4 Goal Translator

```
You are a helpful assistant that converts natural language goals into symbolic goals. The
symbolic goal must only use available predicates.
It is fine if the natural language goal can not be fully captured by the available
predicates, please just output the symbolic goal that is as close as possible to the natural
language goal.

Example:
Available entities: ['plate']
Available predicates: {'obj_in_sink(a)': 'check whether object a is in sink or not'}
Goal: Wash the plate and put it in sink.
Output:
{
    "Symbolic goal": ["obj_in_sink(plate)"]
}

{domain_desc}
Available entities: {entities}
Available predicates: {predicates}
Goal: {goal_spec}
Output (please don't output ```json):
```

APPENDIX C
DOMAIN DESIGN

We provide further details of the designed domains including: (i) the available objects, (ii) the available primitive actions, (ii) Simple and complex tasks, and (iii) language feedback templates.

C.1 StoreObjects

Available objects:

- A large object, *i.e.*, a shelf or coaster that can not be picked up
- A number of small objects that are to be stored on the large object

Available primitive actions:

- `pick_up(a)`: pick up an object a
- `place_on_table(a)`: place an object a on table
- `place_first_on_second(a,b)`: place an object a on object b

Simple and complex tasks:

- Simple task 1: stack a small object a on the large object b
- Simple task 2: stack a small object a on a small object b
- Simple task 3: put a small object a on table
- Complex task: store objects on the large object a following the order: b on a, c on b, ...

Language feedback templates:

- Explain infeasible action
  - Precondition 1: you can't execute `pick_up(a)` because the gripper is already occupied
  - Precondition 2: you can't execute `pick_up(a)` because there is something on a
  - Precondition 3: you can't execute `pick_up(a)` because a can not be grasped by the gripper as it is too wide
  - Precondition 4: you can't execute `place_on_table(a)` because object a is not held by the gripper
  - Precondition 5: you can't execute `place_first_on_second(a,b)` because object a is not held by the gripper
  - Precondition 6: you can't execute `place_first_on_second(a,b)` because there is something on b

- Explain unsatisfied goal
  - Unsatisfied goal 1: you haven't achieved the goal because object a is not yet on b
  - Unsatisfied goal 2: you haven't achieved the goal because object a is not on table

C.2 SetTable

Available objects:

- A table mat that can not be moved
- A plate that can only be pushed but not grasped
- A number of small objects that are to be placed on plate / table mat

Available primitive actions:

- `pick_up(a)`: pick up an object a
- `place_on_table(a)`: place an object a on table
- `place_first_on_second(a,b)`: place an object a on object b
- `push_plate_on_object(a,b)`: push a plate a onto object b

Simple and complex tasks:

- Simple task 1: place a small object a on table mat / plate b
- Simple task 2: place the plate a on table mat b
- Simple task 3: place a small object a on table
- Complex task: set a breakfast table with plate b on table mat a, c on b, ...

Language feedback templates:

- Explain infeasible action
  - The 6 precondition explanations as in StoreObjects
  - Precondition 7: you can't execute `push_plate_on_object(a,b)` because object a is not a plate
  - Precondition 8: you can't execute `push_plate_on_object(a,b)` because the gripper is occupied
  - Precondition 9: you can't execute `push_plate_on_object(a,b)` because there is something on a
  - Precondition 10: you can't execute `push_plate_on_object(a,b)` because there is something on b
  - Precondition 11: you can't execute `push_plate_on_object(a,b)` because object b is not thin enough as its height is greater than xxx
- Explain unsatisfied goal
  - Unsatisfied goal 1: you haven't achieved the goal because object a is not yet on b
  - Unsatisfied goal 1: you haven't achieved the goal because object a is not on table

C.3 CookMeal

Available objects:

- A heavy pot that can contain food ingredients and water, but can not be moved by the gripper
- A faucet that can get water from with a container
- One or more cups that can be used to get water from facet and contain water
- A number of food ingredients that are to be put into the pot

Available primitive actions:

- `pick_up(a)`: pick up an object a
- `place_on_table(a)`: place an object a on table
- `place_first_in_second(a,b)`: place an object a into container b
- `get_water_from_faucet(a)`: get water from the faucet with cup a
- `pour_water_from_first_to_second(a,b)`: pour water from container a to container b

Simple and complex tasks:

- Simple task 1: pour water into cup a and put it on table
- Simple task 2: pour water into pot a
- Simple task 3: put object a into pot b
- Complex task: pour water and put a, b, ... in pot c, pour water into cup d and put it on table

Language feedback templates:

- Explain infeasible action
  - Precondition 1: you can't execute `pick_up(a)` because the gripper is already occupied
  - Precondition 2: you can't execute `pick_up(a)` because a is in a container
  - Precondition 3: you can't execute `pick_up(a)` because a can not be grasped by the gripper as it is too wide
  - Precondition 4: you can't execute `place_on_table(a)` because object a is not held by the gripper
  - Precondition 5: you can't execute `place_first_in_second(a,b)` because object a is not held by the gripper
  - Precondition 6: you can't execute `place_first_in_second(a,b)` because object b is not a container, only objects with category cup, pot, and basket are containers
  - Precondition 7: you can't execute `place_first_in_second(a,b)` because object a is not food
  - Precondition 8: you can't execute `place_first_in_second(a,b)` because object b can not contain food as it's too small, i.e., its width is smaller than 10
  - Precondition 9: you can't execute `get_water_from_faucet(a)` because object a is not held by the gripper
  - Precondition 10: you can't execute `get_water_from_faucet(a)` because object a is not a container
  - Precondition 11: you can't execute `get_water_from_faucet(a)` because object a already contains water
  - Precondition 12: you can't execute `pour_water_from_first_to_second(a,b)` because object a is not held by the gripper
  - Precondition 13: you can't execute `pour_water_from_first_to_second(a,b)` because object a does not have water
  - Precondition 14: you can't execute `pour_water_from_first_to_second(a,b)` because object b is not a container
- Explain unsatisfied goal
  - Unsatisfied goal 1: you haven't achieved the goal because object a does not have water
  - Unsatisfied goal 2: you haven't achieved the goal because object a is not on table
  - Unsatisfied goal 3: you haven't achieved the goal because object a is not in pot b

APPENDIX D
EXAMPLES OF LEARNED PREDICATES AND OPERATORS

We present examples of learned predicates and operators in the real-world SetTable domain. Note that these predicates and operators are pretty much a superset of those in the real-world StoreObjects domain. We show the predicates as Python functions and the operators in a PDDL domain file. For the learned predicates and operators in the simulated domains, please refer to our github repo.

```python
import numpy as np
from typing import *
from real_robot.perception_api import (
    get_detected_object_list,
    get_object_category,
    get_object_center_3d,
    get_object_size_3d,
    get_gripper_position_3d,
    get_gripper_max_open_width,
    get_gripper_open_width,
    get_table_height,
)

# All numbers are in meters. Please reason about the proper tolerance value for each predicate to
incoporate real-world perception uncertainty. You may design different tolerance for different variables,
like along different axis.
# Usually, when objects are close to each other (e.g., stacked together), the segmented objects may
include part of each other and their bounding boxes may overlap. You need to handle this using the
tolerance properly.

# Utility functions:
# <utility>
def get_object_x_range(a) -> np.ndarray:
    """
    Get the range of object a along x axis
    :param a: string, name of detected object
    """
    center = get_object_center_3d(a)
    extent = get_object_size_3d(a)
```

```
    return np.array([center[0] - extent[0] / 2, center[0] + extent[0] / 2])
# <end-of-utility>

# <utility>
def get_object_y_range(a) -> np.ndarray:
    """
    Get the range of object a along y axis
    :param a: string, name of detected object
    """
    center = get_object_center_3d(a)
    extent = get_object_size_3d(a)
    return np.array([center[1] - extent[1] / 2, center[1] + extent[1] / 2])
# <end-of-utility>

# Predicates:
# <predicate>
def obj_in_gripper(a) -> bool:
    """
    Description: <<whether object a is held by the gripper>>
    The predicate holds True when gripper is half open and object a is close to the gripper.
    :param a: string, name of detected object
    """
    gripper_open_tol = 0.01
    z_tol = 0.02
    xy_tol = 0.01

    # check whether gripper is half-open
    # gripper_width < max_gripper_width + gripper_open_tol
    if get_gripper_open_width() < get_gripper_max_open_width() - gripper_open_tol:
        gripper_position = get_gripper_position_3d()
        object_center = get_object_center_3d(a)
        object_extent = get_object_size_3d(a)
        # check whether the distance between object a and gripper along z is within z_tol
        # abs(gripper_z - object_center_z) < z_tol
        if np.abs(gripper_position[2] - object_center[2]) < z_tol:
            # check whether the distance between object a and gripper along x and y is within half the
            extent of a
            # abs(gripper_xy - object_center_xy) < object_extent_xy / 2 + xy_tol
            if np.all(np.abs(gripper_position[:2] - object_center[:2]) < object_extent[:2] / 2 + xy_tol):
                return True
            else:
                False
        else:
            False
    else:
        return False
# <end-of-predicate>

# <predicate>
def obj_on_obj(a: str, b: str) -> bool:
    """
    Description: <<check whether object a is on top of object b or not>>
    The predicate holds True if the bottom of object a is within a certain tolerance above the top of
    object b,
    and their x and y projections overlap.
    :param a: string, name of detected object
    :param b: string, name of detected object
    """
    z_tol = 0.01  # Reduced tolerance for the z-axis to consider object a is on top of object b
    overlap_tol = 0.01  # Reduced tolerance for the overlap in x and y axis

    # Get the center and size of both objects
    center_a, size_a = get_object_center_3d(a), get_object_size_3d(a)
    center_b, size_b = get_object_center_3d(b), get_object_size_3d(b)

    # Calculate the z position of the bottom of object a and the top of object b
    bottom_a = center_a[2] - size_a[2] / 2
    top_b = center_b[2] + size_b[2] / 2

    # Check if bottom of a is within tolerance above top of b
    if not (bottom_a <= top_b + z_tol and bottom_a >= top_b - z_tol):
        return False
```

```
    # Check if the projections of a and b on the x and y axes overlap
    x_range_a = get_object_x_range(a)
    x_range_b = get_object_x_range(b)
    y_range_a = get_object_y_range(a)
    y_range_b = get_object_y_range(b)

    # Check overlap in x and y axis
    overlap_x = min(x_range_a[1], x_range_b[1]) - max(x_range_a[0], x_range_b[0]) > -overlap_tol
    overlap_y = min(y_range_a[1], y_range_b[1]) - max(y_range_a[0], y_range_b[0]) > -overlap_tol

    return overlap_x and overlap_y
# <end-of-predicate>

# <predicate>
def gripper_empty() -> bool:
    """
    Description: <<check whether the gripper is empty, the predicate holds when the gripper is not
    holding any object>>
    """
    gripper_open_tol = 0.01  # Tolerance for considering the gripper as not fully open
    max_gripper_width = get_gripper_max_open_width()
    current_gripper_width = get_gripper_open_width()

    # If the gripper is almost fully open, we consider it empty
    if current_gripper_width >= max_gripper_width - gripper_open_tol:
        return True

    # If the gripper is not fully open, check proximity of objects to gripper
    detected_objects = get_detected_object_list()
    for obj in detected_objects:
        if obj_in_gripper(obj):
            return False  # Found an object in the gripper

    return True  # No object found in the gripper
# <end-of-predicate>

# <predicate>
def obj_on_table(a: str) -> bool:
    """
    Description: <<check whether object a is on the table or not>>
    The predicate holds True if the bottom of object a is within a certain tolerance above the table
    height.
    :param a: string, name of detected object
    """
    z_tol = 0.01  # Tolerance for considering object a is on the table

    # If object a is on another object, we consider it not on table
    detected_objects = get_detected_object_list()
    for obj in detected_objects:
        if obj == a:
            continue # Skip the object itself
        if obj_on_obj(a, obj):
            return False # Found object a on top of obj

    # Get the table height
    table_height = get_table_height()

    # Get the center and size of object a
    center_a, size_a = get_object_center_3d(a), get_object_size_3d(a)

    # Calculate the z position of the bottom of object a
    bottom_a = center_a[2] - size_a[2] / 2

    # Check if bottom of a is within tolerance above the table height
    return (bottom_a <= table_height + z_tol) and (bottom_a >= table_height - z_tol)
# <end-of-predicate>

# <predicate>
def obj_too_large_to_grasp(a: str) -> bool:
    """
    Description: <<check whether object a is too large for the gripper to grasp, the predicate holds when
    the object's size exceeds the gripper's grasping capacity>>
    :param a: string, name of detected object
```

```
    """
    # Get the maximum open width of the gripper
    max_gripper_width = get_gripper_max_open_width()

    # Get the size of object a
    size_a = get_object_size_3d(a)

    # Check if the object's size along the x axis (gripper open direction) exceeds the gripper's grasping
    capacity
    return size_a[0] > max_gripper_width
# <end-of-predicate>

# <predicate>
def obj_free_of_objects(a: str) -> bool:
    """
    Description: <<check whether object a does not have any other objects on top of it, the predicate
    holds when there are no objects placed on object a>>
    :param a: string, name of detected object
    """
    detected_objects = get_detected_object_list()
    for obj in detected_objects:
        if obj == a:
            continue  # Skip the object itself
        if obj_on_obj(obj, a):
            return False  # Found an object on top of object a
    return True  # No objects found on top of object a
# <end-of-predicate>

# <predicate>
def is_plate(a) -> bool:
    """
    Description: <<check whether object a is a plate, the predicate holds true if a is a plate>>
    :param a: string, name of detected object a
    """
    # Get the category of object a
    object_a_category = get_object_category(a)

    # Check whether the category of object a is plate
    if object_a_category == "plate":
        return True
    else:
        return False
# <end-of-predicate>

# <predicate>
def obj_thin_enough(a) -> bool:
    """
    Description: <<check whether object a is thin enough, the predicate holds true if the height of
    object a is less than or equal to 0.01>>
    :param a: string, name of detected object a
    """
    # Get the size of object a
    size_a = get_object_size_3d(a)

    # check whether the height of object a is less than or equal to 0.01
    if size_a[2] <= 0.01:
        return True
    else:
        return False
# <end-of-predicate>
```

Listing 1: Learned Predicates for Real-world SetTable Domain

```
(define (domain set_table)
  (:requirements :typing)
  (:types default)

  (:predicates (obj_in_gripper ?v0 - default)
        (not_obj_in_gripper ?v0 - default)
        (obj_on_obj ?v0 - default ?v1 - default)
        (not_obj_on_obj ?v0 - default ?v1 - default)
```

```
        (gripper_empty)
        (not_gripper_empty)
        (obj_on_table ?v0 - default)
        (not_obj_on_table ?v0 - default)
        (obj_too_large_to_grasp ?v0 - default)
        (not_obj_too_large_to_grasp ?v0 - default)
        (obj_free_of_objects ?v0 - default)
        (not_obj_free_of_objects ?v0 - default)
        (is_plate ? v0 - default)
        (not_is_plate ? v0 - default)
        (obj_thin_enough ? v0 - default)
        (not_obj_thin_enough ? v0 - default)
        (pick_up ?v0 - default)
        (place_on_table ?v0 - default)
        (place_first_on_second ?v0 - default ?v1 - default)
)
; (:actions pick_up place_on_table place_first_on_second push_plate_on_object)



        (:action place_on_table_1
                :parameters (?v_0 - default)
                :precondition (and (obj_in_gripper ?v_0)
                        (place_on_table ?v_0))
                :effect (and
                        (not_obj_in_gripper ?v_0)
                        (gripper_empty)
                        (not (obj_in_gripper ?v_0))
                        (obj_on_table ?v_0))
        )


        (:action pick_up_1
                :parameters (?v_1 - default ?v_0 - default)
                :precondition (and (gripper_empty)
                        (obj_free_of_objects ?v_0)
                        (not_obj_too_large_to_grasp ?v_0)
                        (pick_up ?v_0)
                        (obj_on_obj ?v0 ?v_1))
                :effect (and
                        (obj_in_gripper ?v_0)
                        (not (obj_on_obj ?v_0 ?v_1))
                        (obj_free_of_objects ?v_1)
                        (not (gripper_empty))
                        (not (not_obj_in_gripper ?v_0)))
        )


        (:action pick_up_2
                :parameters (?v_0 - default)
                :precondition (and (obj_on_table ?v_0)
                        (gripper_empty)
                        (obj_free_of_objects ?v_0)
                        (not_obj_too_large_to_grasp ?v_0)
                        (pick_up ?v_0))
                :effect (and
                        (not (gripper_empty))
                        (not (not_obj_in_gripper ?v_0))
                        (obj_in_gripper ?v_0)
                        (not (obj_on_table ?v_0)))
        )


        (:action place_first_on_second_1
                :parameters (?v_1 - default ?v_0 - default)
                :precondition (and (obj_in_gripper ?v_0)
                        (obj_free_of_objects ?v_1)
                        (place_first_on_second ?v_0 ?v_1))
                :effect (and
                        (not_obj_in_gripper ?v_0)
                        (not (obj_free_of_objects ?v_1))
                        (gripper_empty)
                        (obj_on_obj ?v_0 ?v_1)
```

```
                        (not (obj_in_gripper ?v_0)))
        )


        (:action push_plate_on_object_1
                :parameters (?v_1 - default ?v_0 - default)
                :precondition (and (gripper_empty)
                        (obj_free_of_objects ?v_0)
                        (obj_free_of_objects ?v_1))
                        (obj_thin_enough ?v_1)
                        (is_plate ?v_0)
                        (push_plate_on_object ?v_0 ?v_1))
                :effect (and
                        (obj_on_obj ?v_0 ?v_1)
                        (not (obj_free_of_objects ?v_1)))
        )



)
```

Listing 2: Learned Operators for Real-world SetTable Domain