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) [1, 2, 3]. In this work, we implement a vairant of the Cluster-and-search operator learning algorithm [1] 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 [1]: (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 [4], 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 [5] 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 [6], 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

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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