LEARNING ADAPTIVE PLANNING REPRESENTATIONS WITH NATURAL LANGUAGE GUIDANCE

Lionel Wong[1*] **Jiayuan Mao**[1*] **Pratyusha Sharma**[1*] **Zachary S. Siegel**[2] **Jiahai Feng**[3]
Noa Korneev[4] **Joshua B. Tenenbaum**[1] **Jacob Andreas**[1]
[1]MIT [2]Princeton University [3]UC Berkeley [4]Microsoft

ABSTRACT

Effective planning in the real world requires not only world knowledge, but the ability to leverage that knowledge to build the *right representation* of the task at hand. Decades of hierarchical planning techniques have used domain-specific temporal *action abstractions* to support efficient and accurate planning, almost always relying on human priors and domain knowledge to decompose hard tasks into smaller subproblems appropriate for a goal or set of goals. This paper describes *Ada* (Action Domain Acquisition), a framework for automatically constructing task-specific planning representations using task-general background knowledge from language models (LMs). Starting with a general-purpose hierarchical planner and a low-level goal-conditioned policy, Ada **interactively learns a library of planner-compatible high-level action abstractions and low-level controllers adapted to a particular domain of planning tasks**. On two language-guided interactive planning benchmarks (*Mini Minecraft* and *ALFRED Household Tasks*), Ada strongly outperforms other approaches that use LMs for sequential decision-making, offering more accurate plans and better generalization to complex tasks.

1 INTRODUCTION

People make complex plans over long timescales, flexibly adapting what we *know* about the world in general to govern how we act in specific situations. To make breakfast in the morning, we might convert a broad knowledge of cooking and kitchens into tens of fine-grained motor actions in order to find, crack, and fry a specific egg; to achieve a complex research objective, we might plan a routine over days or weeks that begins with the low-level actions necessary to ride the subway to work. The problem of *adapting general world knowledge to support flexible long-term planning* is one of the unifying challenges of AI. While decades of research have developed representations and algorithms for solving restricted and shorter-term planning problems, generalized and long-horizon planning remains a core, outstanding challenge for essentially all AI paradigms, including classical planning (Erol et al., 1994), reinforcement learning (Sutton et al., 1999), and modern generative AI (Wang et al., 2023a).

How do humans solve this computational challenge? A growing body of work in cognitive science suggests that people come up with *hierarchical, problem-specific representations* of their actions and environment to suit their goals, tailoring how they represent, remember, and reason about the world to plan efficiently for a particular set of tasks (e.g., Ho et al., 2022). In AI, a large body of work has studied *hierarchical planning using domain-specific temporal abstractions*—progressively decomposing high-level goals into sequences abstract actions that eventually bottom out in low-level control. An extensive body of work has explored how to plan using these hierarchical action spaces, including robotic task-and-motion planning (TAMP) systems (Garrett et al., 2021) and hierarchical RL frameworks (Sutton et al., 1999).

However, identifying a set of abstract actions that are relevant and useful for achieving any given set of goals remains the central bottleneck in general. Intuitively, "useful" high-level actions must satisfy many different criteria: they should enable time-efficient high-level planning, correspond feasible low-level action sequences, and compose and generalize to new tasks. Despite efforts to learn high-level actions automatically in both classical planning (Nejati et al., 2006) and RL formulations (Dietterich, 2000), most state-of-the-art robotics and planning systems rely on human expertise to hand-engineer new planning representations for each new domain (Ahn et al., 2022).

[*]Asterisk indicates equal contribution. Correspondence to zyzzyva@mit.edu. Code for this paper will be released at: https://github.com/CatherineWong/llm-operators

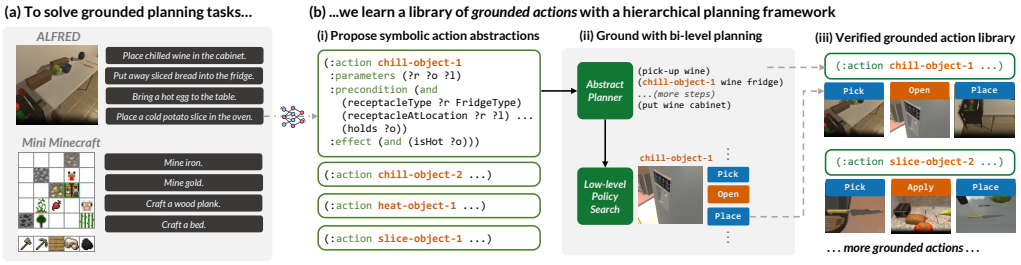

Figure 1: We solve complex planning tasks specified in language and grounded in interactive environments by jointly learning a *library of symbolic high-level action abstractions and modular low-level controllers* associated with each abstraction. Our system leverages background information in language as a prior to *propose useful action abstractions*, then uses a *hierarchical planning framework* to verify and ground them.

In this paper, we introduce *Action Domain Acquisition* (*Ada*), a framework for using background knowledge from language (conveyed via language models) as an initial source of task-relevant domain knowledge. Ada uses language models (LMs) in an interactive planning loop to assemble a *library of composable, hierarchical actions tailored to a given environment and task space*. Each action consists of two components: (1) a *high-level abstraction* represented as a symbolic planning *operator* (Fikes & Nilsson, 1971) that specifies preconditions and action effects as sets of predicates; and (2) a *low-level controller* that can achieve the action's effects by predicting a sequence of low-level actions with a neural network or local search procedure. We study planning in a multitask reinforcement learning framework, in which agents interact with their environments to must solve collections of tasks of varying complexity. Through interaction, Ada incrementally builds a library of actions, ensuring at each step that learned high-level actions compose to produce valid abstract plans and realizable low-level trajectories.

We evaluate Ada (Fig. 1) on two benchmarks, *Mini Minecraft* and *ALFRED* (Shridhar et al., 2020). We compare this approach against three baselines that leverage LMs for sequential decision-making in other ways: to parse linguistic goals into formal specifications that are solved directly by a planner (as in Liu et al. (2023)), to directly predict sequences of high-level subgoals (as in Ahn et al. (2022)), and to predict libraries of actions defined in general imperative code (as in Wang et al. (2023a)). In both domains, we show that Ada learns action abstractions that allow it to solve dramatically more tasks on each benchmark than these baselines, and that these abstractions compose to enable efficient and accurate planning in complex, unseen tasks.

2 Problem Formulation

We assume access to an environment $\langle \mathcal{X}, \mathcal{U}, \mathcal{T} \rangle$, where $\mathcal{X}$ is the (raw) state space, $\mathcal{U}$ is the (low-level) action space (e.g., robot commands), and $\mathcal{T}$ is a deterministic transition function $\mathcal{T} : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$. We also have a set of features (or "predicates") $\mathcal{P}$ that define an abstract state space $\mathcal{S}$: each abstract state $s \in \mathcal{S}$ is composed of a set of objects and their features. For example, a simple scene that contains bread on a table could be encoded as an abstract state with two objects *A* and *B*, and atoms $\{\textit{bread}(\mathrm{A}), \textit{table}(\mathrm{B}), \textit{on}(\mathrm{A}, \mathrm{B})\}$. We assume the mapping from environmental states to abstract states $\Phi : \mathcal{X} \to \mathcal{S}$ is given and fixed (though see Migimatsu & Bohg, 2022 for how it might be learned).

In addition to the environment, we have a collection of tasks t. Each t is described by a natural language instruction ℓ_t, corresponding to a goal predicate (which is not directly observed). In this paper, we assume that predicates may be defined in terms of abstract states, i.e., $g_t : \mathcal{S} \to \{T, F\}$. Our goal is to build an agent that, given the initial state $x_0 \in \mathcal{X}$ and the natural language instruction ℓ_t, can generate a sequence of low-level actions $\{u_1, u_2, \cdots, u_H\} \in \mathcal{U}^H$ such that $g_t(\Phi(x_H))$ is true (where x_H is the terminal state of sequentially applying $\{u_i\}$ on x_0). The agent receives reward signal only upon achieving the goal specified by g_t.

Given a very large number of interactions, a sufficiently expressive reflex policy could, in principle, learn a policy that maps from low-level states to low-level actions conditioned on the language instruction $\pi(u \mid x; \ell_t)$. However, for very long horizons H and large state spaces (e.g., composed of many objects and compositional goals), such algorithms can be highly inefficient or effectively infeasible. The key idea behind our approach is to use natural language descriptions ℓ_t to bootstrap a high-level action space $\mathcal{A}$ over the abstract state space $\mathcal{S}$ to accelerate learning and planning.

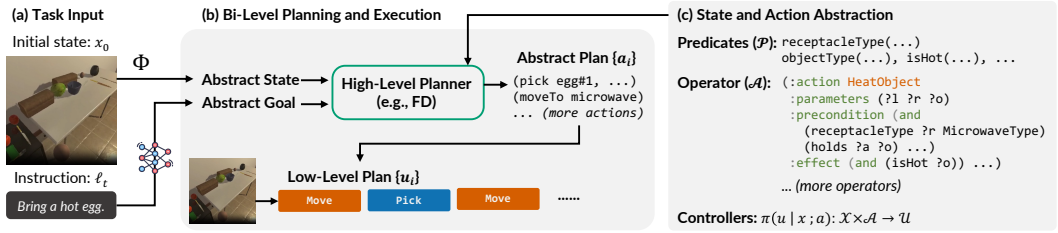

Figure 2: Representation for our (a) task input, (b) the bi-level planning and execution pipeline for inference time, and (c) the abstract state and action representation.

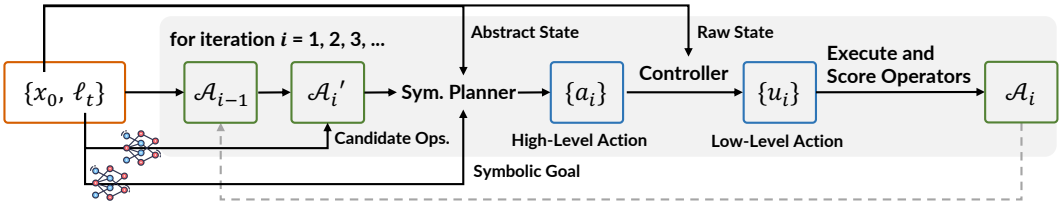

Figure 3: The overall framework. Given task environment states and descriptions, at each iteration, we first propose candidate abstract actions (operators) $\mathcal{A}_i'$, then uses bi-level planning and execution to solve tasks. We add operators to the operator library based on the execution result.

Formally, our approach learns a library of high-level actions (operators) $\mathcal{A}$. As illustrated in Fig. 2b, each $a \in \mathcal{A}$ is a tuple of $\langle name, args, pre, eff, controller \rangle$. *name* is the name of the action, *args* is a list of variables, usually denoted by $?x, ?y, etc.$, *pre* is a precondition formula based on the variables *args* and the features $\mathcal{P}$, and *eff* is the effect, which is also defined in terms of *args* and $\mathcal{P}$. Finally, $controller : \mathcal{X} \to \mathcal{U}$ is a low-level policy associated with the action. The semantics of the preconditions and effects is: for any state x such that $pre(\Phi(x))$, executing *controller* starting in x (for an indefinite number of steps) will yield a state x' such that $eff(\Phi(x'))$ (Lifschitz, 1986). *In this framework, $\mathcal{A}$ defines a partial, abstract world model of the underlying state space.*

As shown in Fig. 2b, given the set of high-level actions and a parse of the instruction ℓ_t into a first-order logic formula, we can leverage symbolic planners (e.g., Helmert, 2006) to first compute a high-level plan $\{a_1, \cdots, a_K\} \in \mathcal{A}^K$ that achieves the goal ℓ_t symbolically, and then refine the high-level plan into a low-level plan with the action controllers. This bi-level planning approach decomposes long-horizon planning problems into several short-horizon problems. Furthermore, it can also leverage the compositionality of high-level actions $\mathcal{A}$ to generalize to longer plans.

3 Action Abstractions from Language

As illustrated in Fig. 3, our framework, *Action Domain Acquisition* (*Ada*) learns action abstractions iteratively as it attempts to solve tasks. Our algorithm is given a dataset of tasks and their corresponding language descriptions, the feature set $\mathcal{P}$, and optionally an initial set of high-level action operators $\mathcal{A}_0$. At each iteration i, we first use a large language model (LLM) to propose a set of novel high-level action definitions $\mathcal{A}_i'$ based on the features $\mathcal{P}$ and the language goals $\{\ell_t\}$ (Section 3.1). Next, we use a LLM to also translate each language instruction ℓ_t into a symbolic goal description F_t, and use a bi-level planner to compute a low-level plan to accomplish ℓ_t (Section 3.2). Then, based on the planning and execution results, we score each operator in $\mathcal{A}_i$ and add ones to the verified library if they have yielded successful execution results (Section 3.4). To accelerate low-level planning, we simultaneously learn local subgoal-conditioned policies (i.e., the controllers for each operator; Section 3.3). Algorithm 1 summarizes the overall framework.

A core goal of our approach is to adapt the initial action abstractions proposed from an LLM prior into a set of *useful* operators $\mathcal{A}*$ that permit efficient and accurate planning on a dataset of tasks and ideally, that generalize to future tasks. While language provides a key initial prior, our formulation refines and verifies the operator library to adapt to a given planning procedure and environment (similar to other action-learning formulations like Silver et al., 2021). Our formulation ensures not only that the learned operators respect the dynamics of the environment, but also fit their grain of abstraction according to the capacity of the controller, trading off between fast high-level planning and efficient low-level control conditioned on each abstraction.

Algorithm 1 Action Abstraction Learning from Language

Input: Dataset of tasks and their language descriptions $\{\ell_t\}$
Input: Predicate set $\mathcal{P}$
Input: Optionally, an initial set of abstract operators $\mathcal{A}_0$, or $\mathcal{A}_0 = \emptyset$
1: Initialize subgoal-conditioned policy π_θ.
2: **for** $i = 1, 2, \cdots, M$ **do**
3: $\quad \mathcal{A}_i \leftarrow \mathcal{A}_{i-1} \cup \text{ProposeOperatorDefinitions}(\mathcal{P}, \{\ell_t\})$ ▷ Section 3.1
4: $\quad$ **for** each unsolved task j: $(x_0^{(j)}, \ell_t^{(j)})$ **do**
5: $\quad\quad \bar{u} \leftarrow \text{BiLevelPlan}(\mathcal{A}_i, \ell_t^{(j)}, \pi)$ ▷ Section 3.2
6: $\quad\quad \textit{result}^{(j)} \leftarrow \text{Execute}(x_0^{(j)}, \bar{u})$ ▷ Execute the plan
7: $\quad \theta \leftarrow \text{UpdateSubgoalPolicy}(\theta, \textit{result})$ ▷ Section 3.3
8: $\quad \mathcal{A}_i \leftarrow \text{ScoreAndFilter}(\mathcal{A}_i, \textit{result})$ ▷ Section 3.4
return $\mathcal{A}_M$

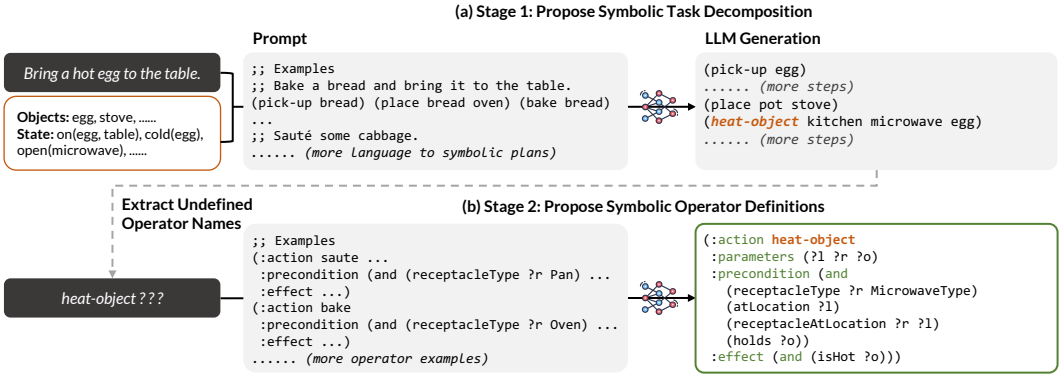

Figure 4: Our two-stage prompting method for generating candidate operator definitions. (a) Given a task instruction, we first prompt an LLM to generate a candidate symbolic task decomposition. (b) We then extract undefined operator names that appear in the sequences and prompt an LLM to generate symbolic definitions.

3.1 Operator Proposal: $\mathcal{A}_i \leftarrow \mathcal{A}_{i-1} \cup \text{ProposeOperatorDefinitions}(\mathcal{P}, \{\ell_t\})$

At each iteration i, we use a pretrained LLM to extend the previous operator library $\mathcal{A}_{i-1}$ with a large set of candidate operator definitions proposed by the LLM based on the task language descriptions and environment features $\mathcal{P}$. This yields an extended candidate library $\mathcal{A}_i'$ where each $a \in \mathcal{A}_i' = \langle \textit{name}, \textit{args}, \textit{pre}, \textit{eff} \rangle$ where *name* is a human-readable action name and *args*, *pre*, *eff* are a PDDL operator definition. We employ a two-stage prompting strategy: symbolic task decomposition followed by symbolic operator definition.

Example. Fig. 4 shows a concrete example. Given a task instruction (*Bring a hot egg to the table*) and the abstract state description, we first prompt the LLM to generate an abstract task decomposition, which may contain operator names that are undefined in the current operator library. Next, we extract the names of those undefined operators and prompt LLMs to generate the actual symbolic operator descriptions, in this case, the new *heat-object* operator.

Symbolic task decomposition. For a given task ℓ_t and a initial state x_0, we first translate the raw state x_0 into a symbolic description $\Phi(x_0)$. To constrain the length of the state description, we only include unary features in the abstract state (i.e., only object categories and properties). Subsequently, we present a few-shot prompt to the LLM and query it to generate a proposed task decomposition conditioned on the language description ℓ_t. It generates a sequence of named high-level actions and their arguments, which explicitly can include high-level actions that are not yet defined in the current action library. We then extract all the operator names proposed across tasks as the candidate high-level operators. Note that while in principle we might use the LLM-proposed task decomposition itself as a high-level plan, we find empirically that this is less accurate and efficient than a formal planner.

Symbolic operator definition. With the proposed operator names and their usage examples (i.e., the actions and their arguments in the proposed plans), we then few-shot prompt the LLM to generate candidate operator *definitions* in the PDDL format (argument types, and pre/postconditions defined based on features in $\mathcal{P}$). We also post-process the generated operator definitions to remove feature

names not present in $\mathcal{P}$ and correct syntactic errors. We describe implementation details for our syntax correction strategy in the appendix.

3.2 Goal Proposal and Planning: $result^{(j)} \leftarrow \text{Execute}(x_0^{(j)}, \text{BiLevelPlan}(\mathcal{A}_i, \ell_t^{(j)}, \pi))$

At each iteration i, we then attempt to *BiLevelPlan* for unsolved tasks in the dataset. This step attempts to find and execute a low-level action sequence $\{u_1, u_2, \cdots, u_H\} \in \mathcal{U}^H$ for each task using the proposed operators in $\mathcal{A}'_i$ that satisfies the unknown goal predicate g_t for each task. This provides the environment reward signal for action learning. Our *BiLevelPlan* has three steps.

Symbolic goal proposal: As defined in Sec. 2, each task is associated with a queryable but unknown goal predicate g_t that can be represented as a first-order logic formula f_t over symbolic features in $\mathcal{P}$. Our agent only has access to a linguistic task description ℓ_t, so we use a few-shot prompted LLM to predict candidate goal formulas F'_t conditioned on ℓ_t and features $\mathcal{P}$.

High-level planning: Given each candidate goal formula $f'_t \in F'_t$, the initial abstract problem state s_0, and the current candidate operator library $\mathcal{A}'$, we search for a *high-level plan* $P_A = \{(a_1, o_{1_i}...), \cdots, (a_K, o_{K_i}...)\}$ as a sequence of high-level actions from $\mathcal{A}'$ concretized with object arguments o, such that executing the action sequence would satisfy f'_t according to the operator definitions. This is a standard symbolic PDDL planning formulation; we use an off-the-shelf symbolic planner, FastDownward (Helmert, 2006) to find high-level plans.

Low-level planning and environment feedback: We then search for a low-level plan as a sequence of low-level actions $\{u_1, u_2, \cdots, u_H\} \in \mathcal{U}^H$, conditioned on the high-level plan structure. Each concretized action tuple $(a_i, o_{1_i}...) \in P_A$ defines a local subgoal sg_i, as the operator postcondition parameterized by the object arguments o. For each $(a_i, o_{1_i}...) \in P_A$, we therefore search for a sequence of low-level actions $u_{i_1}, u_{i_2}...$ that satisfies the local subgoal sg_i. We search with a fixed budget per subgoal, and fail early if we are unable to satisfy the local subgoal sg_i. If we successfully find a complete sequence of low-level actions satisfying all local subgoals sg_i in P_A, we execute all low-level actions and query the hidden goal predicate g_t to determine environment reward. We implement a basic learning procedure to simultaneously learn subgoal-conditioned controllers over time (described in Section 3.3), but our formulation is general and supports many hierarchical planning schemes (such as sampling-based low-level planners (LaValle, 1998) or RL algorithms).

3.3 Low-Level Learning and Guided Search: $\theta \leftarrow \text{UpdateSubgoalPolicy}(\theta, result)$

The sequence of subgoals sg_i corresponding to high-level plans P_A already restricts the local low-level planning horizon. However, we further learn subgoal-conditioned low-level policies $\pi(u|x; sg)$ from environment feedback during training to accelerate low-level planning. To exploit shared structure across subgoals, we learn a shared controller for all operators from $x \in \mathcal{X}$ and conjunctions of predicates in sg. To maximize learning during training, we use a hindsight goal relabeling scheme (Andrychowicz et al., 2017), supervising on all conjunctions of predicates in the state as we roll out low-level search. While the shared controller could be learned as a supervised neural policy, we find that our learned operators sufficiently restrict the search to permit learning an even simpler count-based model from $X, sg \rightarrow u \in \mathcal{U}$. We provide additional details in the Appendix.

3.4 Scoring LLM Operator Proposals: $\mathcal{A}_i \leftarrow \text{ScoreAndFilter}(\mathcal{A}_i, result)$

Finally, we update the learned operator library $\mathcal{A}_i$ to retain candidate operators that were useful and successful in bi-level planning. Concretely, we estimate operator candidate $a'_i \in \mathcal{A}'_i$ accuracy across the bi-level plan executions as s/b where b counts the total times a'_i appeared in a high-level plan and s counts successful execution of the corresponding low-level action sequence to achieve the subgoal associated with a'_i. We retain operators if $b > \tau_b$ and $s/b > \tau_r$, where τ_b, τ_r are hyperparameters. Note that this scoring procedure learns whether operators are accurate and support low-level planning independently of whether the LLM-predicted goals f'_t matched the true unknown goal predicates g_t.

4 Experiments

Domains. We evaluate our approach on two-language specified planning-benchmarks: *Mini Minecraft* and *ALFRED* (Shridhar et al., 2020). *Mini Minecraft* (Fig. 5, *top*) is a procedurally-generated Minecraft-like benchmark (Chen et al., 2021; Luo et al., 2023) on a 2D grid world that requires

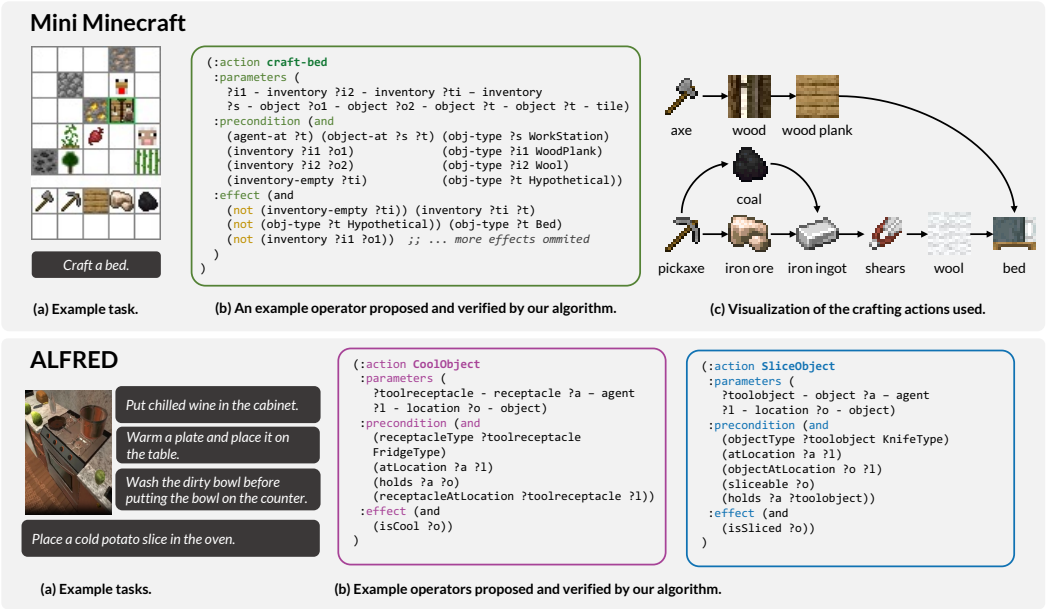

Figure 5: *Top*: (a) The Mini Minecraft environment, showing an intermediate step towards *crafting a bed*. (b) Operator proposed by an LLM and verified by our algorithm through planning and execution. (c) Low-level actions involved in crafting the bed. *Bottom*: (a) The ALFRED household environment. (b) Example operators proposed by LLM and verified by our algorithm, which are composed to solve the *cold potato slice* task.

complex, extended planning. The agent can use tools to mine resources and craft objects. The ability to create new objects that themselves permit new actions yields an enormous action space at each time step (>2000 actions) and very long-horizon tasks (26 high-level steps for the most complex task, without path-planning.) *ALFRED* (Fig. 5, *bottom*) is a household planning benchmark of human-annotated but formally verifiable tasks defined over a simulated Unity environment (Shridhar et al., 2020). The tasks include object rearrangements and those with object states such as heating and cleaning. Ground-truth high-level plans in the ALFRED benchmark compose 5-10 high-level operators, and low-level action trajectories have on average 50 low-level actions. There over 100 objects that the agent can interact with in each interactive environment. See the Appendix for details.

Experimental setup. We evaluate in an iterative continual learning setting; except on the compositional evaluations, we learn from *n=2* iterations through all (randomly ordered) tasks and report final accuracy on those tasks. All experiments and baselines use GPT-3.5. For each task, at each iteration, we sample *n=4* initial goal proposals and *n=4* initial task decompositions, and *n=3* operator definition proposals for each operator name. We report *best-of* accuracy, scoring a task as solved if verification passes on at least one of the proposed goals. For Minecraft, we set the motion planning budget for each subgoal to $\leq$1000 nodes. For ALFRED, which requires a slow Unity simulation, we set it to 50 nodes. Additional temperature and sampling details are in the Appendix.

We evaluate on three *Mini Minecraft* benchmark variations to test how our approach generalizes to complex, compositional goals. In the simplest **Mining** benchmark, all goals involve mining a target item from an appropriate initial resource with an appropriate tool (e.g., Mining *iron* from *iron_ore* with an *axe*). In the harder **Crafting** benchmark, goals involve crafting a target artifact (e.g., a *bed*), which may require mining a few target resources. The most challenging **Compositional** benchmark combines mining and crafting tasks, in environments that only begin with raw resources and two starting tools (axe and pickaxe). Agents may need to compose multiple skills to obtain other downstream resources (see Fig. 5 for an example). To test action generalization, we report evaluation on the *Compositional* using *only* actions learned previously in the **Mining** and **Crafting** benchmarks.

We similarly evaluate on an *ALFRED* benchmark of **Simple and Compositional** tasks drawn from the original task distribution in Shridhar et al. (2020). This distribution contains simple tasks that require picking up an object and placing it in a new location, picking up objects, applying a single household skill to an object and moving them to a new location (e.g., *Put a clean apple on the dining table*), and compositional tasks that require multiple skills (e.g., *Place a hot sliced potato on the*

Mini Minecraft *(n=3)*	**LLM Predicts?**	**Library?**	**Mining**	**Crafting**	**Compositional**
Low-level Planning Only	Goal	✗	31% (σ=0.0%)	9% (σ=0.0%)	9% (σ=0.0%)
Subgoal Prediction	Sub-goals	✗	33% (σ=1.6%)	36% (σ=5.6%)	6% (σ=1.7%)
Code Policy Prediction	Sub-policies	✓	15% (σ=1.2%)	39% (σ=3.2%)	10% (σ=1.7%)
Ada (Ours)	Goal+Operators	✓	100% (σ=0.0%)	100% (σ=7.5%)	100% (σ=4.1%)
ALFRED *(n=3 replications)*	**LLM Predicts?**	**Library?**	**Original (Simple + Compositional Tasks)**		
Low-level Planning Only	Goal	✗	21% (σ=1.0%)		
Subgoal Prediction	Sub-goal	✗	2% (σ=0.4%)		
Code Policy Prediction	Sub-policies	✓	2% (σ=0.9%)		
Ada (Ours)	Goal+Operators	✓	79% (σ=0.9%)		

Table 1: (*Top*) Results on *Mini Minecraft*. Our algorithm successfully recovers all intermediate operators for mining and crafting, which enable generalization to more compositional tasks (which use up to 26 operators) without any additional learning. (*Bottom*) Results on ALFRED. Our algorithm recovers all required household operators, which generalize to more complex compositional tasks. All results report mean performance and STD from *n=3* random replications for all models.

counter). We use a random subset of n=223 tasks, selected from an initial 250 that we manually filter to remove completely misspecified goals (which omit any mention of the target object or skill).

Baselines. We compare our method to three baselines of language-guided planning.

Low-level Planning Only uses an LLM to predict only the symbolic goal specification conditioned on the high-level predicates and linguistic goal, then uses the low-level planner to search directly for actions that satisfy that goal. This baseline implements a model like **LLM+P** (Liu et al., 2023), which uses LLMs to translate linguistic goals into planning-compatible formal specifications, then attempt to plan directly towards these with no additional representation learning.

Subgoal Prediction uses an LLM to predict a sequence of high-level subgoals (as PDDL pre/postconditions with object arguments), conditioned on the high-level predicates, and task goal and initial environment state. This baseline implements a model like **SayCan** (Ahn et al., 2022), which uses LLMs to directly predict goal *and* a sequence of decomposed formal subgoal representations, then applies low-level planning over these formal subgoals.

Code Policy Prediction uses an LLM to predict the definitions of a library of *imperative local code policies* in Python (with cases and control flow) over an imperative API that can query state and execute low-level actions.) Then, as FastDownward planning is no longer applicable, we also use the LLM to predict the function call sequences with arguments for each task. This baseline implements a model like **Voyager** (Wang et al., 2023a), which uses an LLM to predict a library of skills implemented as imperative code for solving individual tasks. Like Voyager, we verify the individual code skills during interactive planning, but do not use a more global learning objective to attempt to learn a concise or non-redundant library.

4.1 Results

What action libraries do we learn? Fig. 5 shows example operators learned on each domain (Appendix A.3 contains the full libraries of operators learned on both domains from a randomly sampled run of the n=3 replications). In *Mini Minecraft*, we manually inspect the library and find that we learn operators that correctly specify the appropriate tools, resources, and outputs for all intermediate mining actions (on **Mining**) and crafting actions (on **Crafting**), allowing perfect direct generalization to the **Compositional** tasks without any additional training on these complex tasks. In *ALFRED*, we compare the learned libraries from all runs to the ground-truth operator library hand-engineered in Shridhar et al. (2020). The ground-truth operator set contains 8 distinct operators corresponding to different compositional skills (e.g., *Slicing*, *Heating*, *Cleaning*, *Cooling*). Across all replications, model reliably recovers semantically identical (same predicate preconditions and postconditions) definitions for *all* of these ground-truth operators, except for a single operator that is defined disjunctively (the ground-truth *Slice* skill specifies either of two types of knives), which we occasionally learn as two distinct operators or only recover with one of these two types.

We also inspect the learning trajectory and find that, through the interactive learning loop, we successfully *reject* many initially proposed operator definitions sampled from the language model that turn out to be redundant (which would make high-level planning inefficient), inaccurate (including apriori reasonable proposals that do not fit the environment specifications, such as proposing to *clean* objects with just a *towel*, when our goal verifiers require washing them with water in a *sink*), or

underspecified (such as those that omit key preconditions, yielding under-decomposed high-level task plans that make low-level planning difficult).

Do these actions support complex planning and generalization? Table 2 shows quantitative results from *n=3* randomly-initialized replications of all models, to account for random noise in sampling from the language model and stochasticity in the underlying environment (ALFRED). On Minecraft, where goal specification is completely clear due to the synthetic language, we solve all tasks in each evaluation variation, including the challenging *Compositional* setting — the action libraries learned from simpler mining/crafting tasks generalize completely to complex tasks that require crafting all intermediate resources and tools from scratch. On ALFRED, we vastly outperform all other baselines, demonstrating that the learned operators are much more effective for planning and compose generalizably to more complex tasks. We qualitatively find that failures on ALFRED occur for several reasons. One is *goal misspecification*, when the LLM does not successfully recover the formal goal predicate (often due to ambiguity in human language), though we find that on average, 92% of the time, the ground truth goal appears as one of the top-4 goals translated by the LLM. We also find failures due to low-level *policy inaccuracy*, when the learned policies fail to account for low-level, often geometric details of the environment (e.g., the learned policies are not sufficiently precise to place a tall bottle on an appropriately tall shelf). More rarely, we see planning failures caused by slight *operator overspecification* (e.g., the *Slice* case discussed above, in which we do not recover the specific disjunction over possible knives that can be used to slice.) Both operator and goal specification errors could be addressed in principal by sampling more (and more diverse) proposals.

How does our approach compare to using the LLM to predict just goals, or predict task sequences? As shown in Table 2, our approach vastly outperforms the **Low-level Planning Only** baseline on both domains, demonstrating the value of the action library for longer horizon planning. We also find a substantial improvement over the **Subgoal Prediction** baseline. While the LLM frequently predicts important high-level aspects of the task subgoal structure (as it does to propose operator definitions), it frequently struggles to robustly sequence these subgoals and predict appropriate concrete object groundings that correctly obey the initial problem conditions or changing environment state. These errors accumulate over the planning horizon, reflected in decreasing accuracy on the compositional Minecraft tasks (on ALFRED, this baseline struggles to solve any more than the basic pick-and-place tasks, as the LLM struggles to predict subgoals that accurately track whether objects are in appliances or whether the agent's single gripper is full with an existing tool.)

How does our approach compare to using the LLM to learn and predict plans using imperative code libraries? Somewhat surprisingly, we find that the *Code Policy* prediction baseline performs unevenly and often very poorly on our benchmarks. (We include additional results in A.2.1 showing that our model also dramatically outperforms this baseline using GPT-4 as the base LLM.) We find several key reasons for the poor performance of this baseline relative to our model, each which validate the key conceptual contributions of our approach. First, the baseline relies on the LLM as the planner – as the skills are written as general Python functions, rather than any planner-specific representation, we do not use an optimized planner like FastDownward. As with *Subgoal Prediction*, we find that the LLM is not a consistent or accurate planner. While it retrieves generally relevant skills from the library for each task, it often struggles to sequence them accurately or predict appropriate arguments given the initial problem state. Second, we find that imperative code is less suited in general as a hierarchical planning representation for these domains than the high-level PDDL and low-level local policy search representation we use in our model. This is because it uses control flow to account for environment details that would otherwise be handled by local search relative to a high-level PDDL action. Finally, our model specifically frames the library learning objective around learning a compact library of skills that enables efficient planning, whereas our Voyager re-implementation (as in Wang et al. (2023a)) simply grows a library of skills which are individually executable and can be used to solve individual, shorter tasks. Empirically, as with the original model in Wang et al. (2023a), this baseline learns *hundreds* of distinct code definitions on these datasets, which makes it harder to accurately plan and generalize to more complex tasks. Taken together, these challenges support our overarching library learning objective for hierarchical planning.

5 Related Work

Planning for language goals. A large body of recent work attempts to use LLMs to solve planning tasks specified in language. One approach is to directly predict action sequences (Huang et al., 2022; Valmeekam et al., 2022; Silver et al., 2022; Wang et al., 2023b), but this has yielded mixed

results as LLMs can struggle to generalize or produce correct plans as problems grow more complex. To combat this, one line of work has explored structured and iterative prompting regimes (e.g., 'chain-of-thought' and feedback) (Mu et al., 2023; Silver et al., 2023; Zhu et al., 2023). Increasingly, other neuro-symbolic work uses LLMs to predict formal goal or action representations that can be verified or solved with symbolic planners (Song et al., 2023; Ahn et al., 2022; Xie et al., 2023; Arora & Kambhampati, 2023). These approaches leverage the benefits of a known planning domain model. Our goal in this paper is to leverage language models to *learn* this domain model. Another line of research aims at using LLMs to generate formal planning domain models for specific problems (Liu et al., 2023) and subsequently uses classical planners to solve the task. However, they are not considering generating grounded or hierarchical actions in an environment and not learning a library of operators that can be reused across different tasks. More broadly, we share the broad goal of building agents that can understand language and execute actions to achieve goals (Tellex et al., 2011; Misra et al., 2017; Nair et al., 2022). See also Luketina et al. (2019) and Tellex et al. (2020).

Learning planning domain and action representations from language. Another group of work has been focusing on learning latent action representations from language (Corona et al., 2021; Andreas et al., 2017; Jiang et al., 2019; Sharma et al., 2022; Luo et al., 2023). Our work differs from them in that we are learning a planning-compatible action abstraction from LLMs, instead of relying on human demonstrations and annotated step-by-step instructions. The more recent Wang et al. (2023a) adopts a similar overall problem specification, to learn libraries of actions as imperative code-based policies. Our results show that learning planning abstractions enables better integration with hierarchical planning, and, as a result, better performance and generalization to more complex problems. Other recent work (Nottingham et al., 2023) learns an environment model from interactive experience, represented as a task dependency graph; we seek to learn a richer state transition model (which represents the effects of actions) decomposed as operators that can be formally composed to verifiably satisfy arbitrarily complex new goals. Guan et al. (2024), published concurrently, seeks to learn PDDL representations; we show how these can be grounded hierarchically.

Language and code. In addition to Wang et al. (2023a), a growing body of work in program synthesis, both by learning lifted program abstractions that compress longer existing or synthesized programs (Bowers et al., 2023; Ellis et al., 2023; Wong et al., 2021; Cao et al., 2023). These approaches (including Wang et al. (2023a)) generally learn libraries defined over imperative and functional programming languages, such as LISP and Python. Our work is closely inspired by these and seeks to learn representations suited specifically to solving long-range planning problems.

Hierarchical planning abstractions. The hierarchical planning knowledge that we learn from LLMs and interactions in the environments are related to hierarchical task networks (Erol et al., 1994; Nejati et al., 2006), hierarchical goal networks (Alford et al., 2016), abstract PDDL domains (Konidaris et al., 2018; Bonet & Geffner, 2020; Chitnis et al., 2021; Asai & Muise, 2020; Mao et al., 2022; 2023), and domain control knowledge (de la Rosa & McIlraith, 2011). Most of these approaches require manually specified hierarchical planning abstractions; others learn them from demonstrations or interactions. By contrast, we leverage human language to guide the learning of such abstractions.

6 Discussion and Future Work

Our evaluations suggest a powerful role for language within AI systems that form complex, long-horizon plans — as a rich source of background knowledge about the right *action abstractions* for everyday planning domains, which contains broad human priors about environments, task decompositions, and potential future goals. A core goal of this paper was to demonstrate how to integrate this knowledge into the search, grounding, and verification toolkits developed in hierarchical planning.

We leave open many possible extensions towards future work. Key **limitations** of our current framework point towards important directions for further integrating LMs and hierarchical planning to scale our approach: here, we build on an existing set of pre-defined symbolic predicates for initially representing the environment state; do not yet tackle fine-grained, geometric motor planning; and use a general LLM (rather than one fine-tuned for extended planning). **Future work** might generally tackle these problems by further asking how else linguistic knowledge and increasingly powerful or multimodal LLMs could be integrated here: to *propose* useful named predicates over initial perceptual inputs (e.g., images) (Migimatsu & Bohg, 2022); or to speed planning by bootstrapping hierarchical planning abstractions using the approach here, but then to progressively transfer planning to another model, including an LLM, to later compose and use the learned representations.

Acknowledgement. We thank anonymous reviewers for their valuable comments. We gratefully acknowledge support from ONR MURI grant N00014-16-1-2007; from the Center for Brain, Minds, and Machines (CBMM, funded by NSF STC award CCF-1231216); from NSF grant 2214177; from NSF grant CCF-2217064 and IIS-2212310; from Air Force Office of Scientific Research (AFOSR) grant FA9550-22-1-0249; from ONR MURI grant N00014-22-1-2740; from ARO grant W911NF-23-1-0034; from the MIT-IBM Watson AI Lab; from the MIT Quest for Intelligence; from Intel; and from the Boston Dynamics Artificial Intelligence Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

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:2204.01691*, 2022. 1, 2, 7, 9

Ron Alford, Vikas Shivashankar, Mark Roberts, Jeremy Frank, and David W Aha. Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *IJCAI*, 2016. 9

Jacob Andreas, Dan Klein, and Sergey Levine. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*, 2017. 9

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. In *NeurIPS*, 2017. 5

Daman Arora and Subbarao Kambhampati. Learning and Leveraging Verifiers to Improve Planning Capabilities of Pre-trained Language Models. *arXiv:2305.17077*, 2023. 9

Masataro Asai and Christian Muise. Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (To Strips). In *IJCAI*, 2020. 9

Blai Bonet and Hector Geffner. Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In *ECAI*, 2020. 9

Matthew Bowers, Theo X Olausson, Lionel Wong, Gabriel Grand, Joshua B Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-Down Synthesis for Library Learning. *PACMPL*, 7(POPL): 1182–1213, 2023. 9

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning Better Abstractions with E-graphs and Anti-unification. *PACMPL*, 7(POPL): 396–424, 2023. 9

Valerie Chen, Abhinav Gupta, and Kenneth Marino. Ask Your Humans: Using Human Instructions to Improve Generalization in Reinforcement Learning. In *ICLR*, 2021. 5, 13

Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. GLiB: Efficient Exploration for Relational Model-Based Reinforcement Learning via Goal-Literal Babbling. In *AAAI*, 2021. 9

Rodolfo Corona, Daniel Fried, Coline Devin, Dan Klein, and Trevor Darrell. Modular Networks for Compositional Instruction Following. In *NAACL-HLT*, 2021. 9

Tomás de la Rosa and Sheila McIlraith. Learning Domain Control Knowledge for TLPlan and Beyond. In *ICAPS 2011 Workshop on Planning and Learning*, 2011. 9

Thomas G Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *JAIR*, 13:227–303, 2000. 1

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. DreamCoder: Growing Generalizable, Interpretable Knowledge with Wake–Sleep Bayesian Program Learning. *Philosophical Transactions of the Royal Society*, 381(2251):20220050, 2023. 9

Kutluhan Erol, James Hendler, and Dana S Nau. HTN Planning: Complexity and Expressivity. In *AAAI*, 1994. 1, 9

Richard E Fikes and Nils J Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.*, 2(3-4):189–208, 1971. 2

Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated Task and Motion Planning. *Ann. Rev. Control Robot. Auton. Syst.*, 4:265–293, 2021. 1

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36, 2024. 9

Malte Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006. 3, 5

Mark K Ho, David Abel, Carlos G Correa, Michael L Littman, Jonathan D Cohen, and Thomas L Griffiths. People Construct Simplified Mental Representations to Plan. *Nature*, 606(7912):129–136, 2022. 1

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *ICML*, 2022. 8

Yiding Jiang, Shixiang Shane Gu, Kevin P Murphy, and Chelsea Finn. Language as an Abstraction for Hierarchical Deep Reinforcement Learning. In *NeurIPS*, 2019. 9

George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *JAIR*, 61:215–289, 2018. 9

Steven LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. *Research Report 9811*, 1998. 5

Vladimir Lifschitz. On the Semantics of STRIPS. In *Workshop on Reasoning about Actions and Plans*, 1986. 3

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:2304.11477*, 2023. 2, 7, 9

Jelena Luketina, Nantas Nardelli, Gregory Farquhar, Jakob Foerster, Jacob Andreas, Edward Grefenstette, Shimon Whiteson, and Tim Rocktäschel. A Survey of Reinforcement Learning Informed by Natural Language. In *IJCAI*, 2019. 9

Zhezheng Luo, Jiayuan Mao, Jiajun Wu, Tomás Lozano-Pérez, Joshua B Tenenbaum, and Leslie Pack Kaelbling. Learning Rational Subgoals from Demonstrations and Instructions. In *AAAI*, 2023. 5, 9, 13

Jiayuan Mao, Tomas Lozano-Perez, Joshua B. Tenenbaum, and Leslie Pack Kaelbing. PDSketch: Integrated Domain Programming, Learning, and Planning. In *NeurIPS*, 2022. 9

Jiayuan Mao, Tomás Lozano-Pérez, Joshua B. Tenenbaum, and Leslie Pack Kaelbling. What planning problems can a relational neural network solve? In *NeurIPS*, 2023. 9

Toki Migimatsu and Jeannette Bohg. Grounding Predicates through Actions, 2022. 2, 9

Dipendra Misra, John Langford, and Yoav Artzi. Mapping Instructions and Visual Observations to Actions with Reinforcement Learning. In *EMNLP*, 2017. 9

Yao Mu, Qinglong Zhang, Mengkang Hu, Wenhai Wang, Mingyu Ding, Jun Jin, Bin Wang, Jifeng Dai, Yu Qiao, and Ping Luo. EmbodiedGPT: Vision-Language Pre-training via Embodied Chain of Thought. *arXiv:2305.15021*, 2023. 9

Suraj Nair, Eric Mitchell, Kevin Chen, Silvio Savarese, and Chelsea Finn. Learning Language-Conditioned Robot Behavior from Offline Data and Crowd-Sourced Annotation. In *CoRL*, 2022. 9

Negin Nejati, Pat Langley, and Tolga Konik. Learning Hierarchical Task Networks by Observation. In *ICML*, 2006. 1, 9

Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hannaneh Hajishirzi, Sameer Singh, and Roy Fox. Do Embodied Agents Dream of Pixelated Sheep: Embodied Decision Making using Language Guided World Modelling. In *ICML*, 2023. 9

Pratyusha Sharma, Antonio Torralba, and Jacob Andreas. Skill Induction and Planning with Latent Language. In *ACL*, 2022. 9

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *CVPR*, 2020. 2, 5, 6, 7, 13

Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning Symbolic Operators for Task and Motion Planning. In *IROS*, 2021. 3

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 Foundation Models for Decision Making Workshop*, 2022. 8

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized Planning in PDDL Domains with Pretrained Large Language Models. *arXiv:2305.11014*, 2023. 9

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 *ICCV*, 2023. 9

Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.*, 112(1-2):181–211, 1999. 1

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In *AAAI*, 2011. 9

Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. Robots That Use Language. *Annual Review of Control, Robotics, & Autonomous Systems*, 3:25–55, 2020. 9

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). In *NeurIPS Foundation Models for Decision Making Workshop*, 2022. 8

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:2305.16291*, 2023a. 1, 2, 7, 8, 9

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents, 2023b. 8

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging Language to Learn Program Abstractions and Search Heuristics. In *ICML*, 2021. 9

Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating Natural Language to Planning Goals with Large-Language Models. *arXiv:2302.05128*, 2023. 9

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory, 2023. 9

A APPENDIX

We will release a complete code repository containing our full algorithm implementation, all baselines, and benchmark tasks. Here, we provide additional details on our implementational choices.

A.1 BENCHMARKS

Mini Minecraft (Fig. 5, *top*) is a procedurally-generated Minecraft-like benchmark (Chen et al., 2021; Luo et al., 2023) that requires complex, extended planning. The environment places an agent on a 2D map containing various resources, tools, and crafting stations. The agent can use appropriate tools to mine new items from raw resources (e.g. use an *axe* to obtain *wood* from *trees*), or collect resources into an inventory to craft new objects (e.g. combining *sticks* and *iron ingots* to craft a *sword*, which itself can be used to obtain *feathers* from a *chicken*). The ability to create new objects that themselves permit new actions yields an enormous action space at each time step (>2000 actions, considering different combinations of items to use) and very long-horizon tasks (26 steps for the most complex task, even without path-planning.) The provided environment predicates allow querying object types and inventory contents. Low-level actions allow the agent to move and apply tools to specific resources. To focus on complex crafting, we provide a low-level *move-to* action to move directly to specified locations. Linguistic goal specifications are synthetically generated from a simple grammar over craftable objects and resources (e.g. *Craft a sword*, *Mine iron ore*).

ALFRED (Fig. 5, *bottom*) is a household planning benchmark of human-annotated but formally verifiable tasks defined over a simulated Unity environment (Shridhar et al., 2020). The interactive environment places an agent in varying 3D layouts, each containing appliances and dozens of household objects. The provided environment includes predicates for querying object types, object and agent locations, and classifiers over object states (eg. whether an object is *hot* or *on*). Low-level actions enable the agent to pick up and place objects, apply tools to other objects, and open, close, and turn on appliances. As specified in Shridhar et al. (2020), ground-truth high-level plans in the ALFRED benchmark compose 5-10 high-level operators, and low-level action trajectories have on average 50 low-level actions. There over 100 objects that the agent can interact with in each interactive environment.

As with Minecraft, we provide a low-level method to move the agent directly to specified locations. While ALFRED is typically used to evaluate detailed instruction following, we focus on a *goal-only* setting that only uses the goal specifications. The human-annotated goals introduce ambiguity, underspecification, and errors with respect to the ground-truth verifiable tasks (eg. people refer to *tables* without specifying if they mean the *side table*, *dining table*, or *desk*; a *light* when there are multiple distinct lamps; or a *cabbage* when they want *lettuce*).

A.2 ADDITIONAL METHODS IMPLEMENTATION DETAILS

A.2.1 LLM PROMPTING

We use `gpt-3.5-turbo-16k` for all experiments and baselines. Here, we describe the contents of the LLM few-shot prompts used in our method in more detail. **Symbolic Task Decomposition** For all unsolved tasks, at each iteration, we sample a set of symbolic task descriptions as a sequence of named high-level actions and their arguments. We construct a few-shot prompt consisting of the following components:

1. A brief natural language header (*;;;; Given natural language goals, predict a sequence of PDDL actions*);
2. A sequence of example (l_t, P_A) tuples containing linguistic goals and example task decompositions. To avoid biasing the language model in advance, we provide example task decompositions for similar, constructed tasks that do not use any of the skills that need to be learned in our two domains.

   For example, on ALFRED, these example task decompositions are for example tasks (*bake a potato and put it in the fridge*, *place a baked, grated apple on top of the dining table*, *place a plate in a full sink.*, and *pick up a laptop and then carry it over to the desk lamp, then restart the desk lamp.*), and our example task decompositions suggest named operators

BakeObject, *GrateObject*, *FillObject*, and *RestartObject*, none of which appear in the actual training set.

3. At iterations > 0, we also provide a sequence of sampled (l_t, P_A) tuples randomly sampled from any solved tasks and their discovered high-level plans. This means that few-shot prompting better represents the true task distribution over successive iterations.

In our experiments, we prompt with temperature=1.0 and draw n=4 task decomposition samples per unsolved task.

Symbolic Operator Definition For all unsolved tasks, at each iteration, we sample proposed operator definitions consisting of *args*, *pre*, *eff* conditioned on all undefined operator names that appear in the proposed task decompositions.

For each operator name, we construct a few-shot prompt consisting of the following components:

1. A brief natural language header (*You are a software engineer who will be writing planning operators in the PDDL planning language. These operators are based on the following PDDL domain definition.*
2. The full set of environment predicates vocabulary of high-level environment predicates $\mathcal{P}$, as well as valid named argument values (eg. object types).
3. A sequence of example *name*, *args*, *pre*, *eff* operator definitions demonstrating the PDDL definition format. As with task decomposition, of course, we do not provide any example operator definitions that we wish to learn from our dataset.
4. At iterations > 0, we include as many possible validated *name*, *args*, *pre*, *eff* operators defined in the current library (including new learned operators). If there are shared patterns between operators, this means that few-shot prompting also better represents the true operator structure over successive iterations.

In our experiments, we prompt with temperature=1.0 and draw n=3 task decomposition samples per unsolved task. However, in our pilot experiments, we actually find that sampling directly from the token probabilities defined by this few-shot prompt does not produce sufficiently diverse definitions for each operator name. We instead directly prompt the LLM to produce up to N distinct operator definitions sequentially.

We find that GPT 3.5 frequently produces syntactically invalid operator proposals – proposed operators often include invent predicates and object types that are not defined in the environment vocabulary, do not obey the predicate typing rules, or do not have the correct number and types of arguments. While this might improve with finetuned or larger LLMs, we instead implement a simple post-processing heuristic to correct operators with syntactic errors, or reject operators altogether: as operator pre and postconditions are represented as conjunctions of predicates, we remove any invalid predicates (predicates that are invented or that specify invalid arguments); we collect all arguments named across the predicates and use the ground truth typing to produce the final *args*, and we reject any operators that have 0 valid postcondition predicates. This post-processing procedure frequently leaves operators underspecified (e.g., the resulting operators now are missing necessary preconditions, which were partially generated but syntactically incorrect in the proposal); we allow our full operator learning algorithm to verify and reject these operators.

Symbolic Goal Proposal Finally, as described in 3.2, we also use an LLM to propose a set of candidate goal definitions as FOL formulas F'_t defined over the environment predicates $\mathcal{P}$ for each task. Our prompting technique is very similar to that used in the rest of our algorithm. For each task, we we construct a few-shot prompt consisting of the following components:

1. A brief natural language header (*You are a software engineer who will be writing goal definitions for a robot in the PDDL planning language.*
2. The full set of environment predicates vocabulary of high-level environment predicates $\mathcal{P}$, as well as valid named argument values (eg. object types).
3. A sequence of example l_t, f_t language and FOL goal formulas. In our experiments, during training, unlike in the previous prompts (where including ground truth operators would solve the learning problem), we *do* sample an initial set of goal definitions from the training

Mini Minecraft	**LLM Predicts?**	**Library?**	**Mining**	**Crafting**	**Compositional**
Code Policy Prediction	Sub-policies	✓	12%	37%	11%
Ours	Goal+Operators	✓	100%	100%	100%
ALFRED *(n=3 replications)*	**LLM Predicts?**	**Library?**	**Original (Simple + Compositional Tasks)**		
Code Policy Prediction	Sub-policies	✓	11%		
Ours	Goal+Operators	✓	70%		

Table 2: **Results with GPT-4 as the LLM backbone**: On both *Mini Minecraft* (*Top*) and ALFRED (*Bottom*), our algorithm recovers all required operators, which generalize to more complex compositional tasks. Switching to GPT-4 does not impact performance trends observed across the *Code as Policies (Voyager)* baseline and our method.

distribution as our initial example supervision. We set supervision to a randomly sampled fraction (0.1) of the training distribution.

4. At iterations > 0, we also include l_t, f_t examples from successfully solved tasks.

In our experiments, we prompt with temperature=1.0 and draw n=4 task decomposition samples per unsolved task. As with the operator proposal, we also find that sampling directly from the token probabilities defined by this few-shot prompt does not produce sufficiently diverse definitions for each linguistic goal to correct for ambiguity in the human language (eg. to define the multiple concrete *Table* types that a person might mean when referring to a *table*). We therefore again instead directly prompt the LLM to produce up to N distinct operator definitions sequentially.

We also post-process proposed goals using the same syntactic criterion to remove invalid predicates in the FOL formula, and reject any empty goals.

A.2.2 Policy Learning and Guided Low-level Search

Concretely, we implement our policy-guided low-level action search as the following. We maintain a dictionary D that maps subgoals (a conjunction of atoms) to a set of candidate low-level action trajectories. When planning for a new subgoal sg, if D contains the trajectory, we prioritize trying candidate low-level trajectories in D. Otherwise, we fall back to a brute-force breadth-first search over all possible action trajectories. To populate D, during the BFS, we compute the difference in the environment state before and after the agent executes any sampled trajectory and the corresponding trajectory t that caused the state change. Here the state difference can be viewed as a subgoal sg achieved by executing t. Rather than directly adding the (sg, t) as a key-value pair to D, we *lift* the trajectory and environment state change by replacing concrete objects in sg and t by variables. Note that we update D with each sampled trajectory in the BFS even if it doesn't achieve the subgoal specified in the BFS search.

When the low-level search receives a subgoal sg, we again lift it by replacing objects with variables, and try to match it with entries in D. If D contains multiple trajectories t for a given subgoal sg, we track how often a given trajectory succeeds for a subgoal and prioritize trajectories with the most successes.

A.3 Experiments

Learned Operator Libraries on Minecraft The following shows the full PDDL domain definition including the initial provided vocabulary of symbolic environment constants and predicates, initial pick and place operators and example operator, and all ensuing learned operators combined from the **Mining** and **Crafting** benchmarks.

```
(define (domain crafting-world-v20230404-teleport)
(:requirements :strips)
(:types
    tile
    object
    inventory
    object-type
)
(:constants
```

```
    Key - object-type
    WorkStation - object-type
    Pickaxe - object-type
    IronOreVein - object-type
    IronOre - object-type
    IronIngot - object-type
    CoalOreVein - object-type
    Coal - object-type
    GoldOreVein - object-type
    GoldOre - object-type
    GoldIngot - object-type
    CobblestoneStash - object-type
    Cobblestone - object-type
    Axe - object-type
    Tree - object-type
    Wood - object-type
    WoodPlank - object-type
    Stick - object-type
    Sword - object-type
    Chicken - object-type
    Feather - object-type
    Arrow - object-type
    Shears - object-type
    Sheep - object-type
    Wool - object-type
    Bed - object-type
    Boat - object-type
    SugarCanePlant - object-type
    SugarCane - object-type
    Paper - object-type
    Bowl - object-type
    PotatoPlant - object-type
    Potato - object-type
    CookedPotato - object-type
    BeetrootCrop - object-type
    Beetroot - object-type
    BeetrootSoup - object-type

    Hypothetical - object-type
    Trash - object-type
)
(:predicates
     (tile-up ?t1 - tile ?t2 - tile)
     (tile-down ?t1 - tile ?t2 - tile)
     (tile-left ?t1 - tile ?t2 - tile)
     (tile-right ?t1 - tile ?t2 - tile)

     (agent-at ?t - tile)
     (object-at ?x - object ?t - tile)
     (inventory-holding ?i - inventory ?x - object)
     (inventory-empty ?i - inventory)

     (object-of-type ?x - object ?ot - object-type)
)

(:action move-to
   :parameters (?t1 - tile ?t2 - tile)
   :precondition (and (agent-at ?t1))
   :effect (and (agent-at ?t2) (not (agent-at ?t1)))
 )
(:action pick-up
   :parameters (?i - inventory ?x - object ?t - tile)
   :precondition (and (agent-at ?t) (object-at ?x ?t) (inventory-empty ?i)
     )
```

```
  :effect (and (inventory-holding ?i ?x) (not (object-at ?x ?t)) (not (inventory-empty ?i)))
 )
(:action place-down
  :parameters (?i - inventory ?x - object ?t - tile)
  :precondition (and (agent-at ?t) (inventory-holding ?i ?x))
  :effect (and (object-at ?x ?t) (not (inventory-holding ?i ?x)) (inventory-empty ?i))
 )
(:action mine-iron-ore
  :parameters (?toolinv - inventory ?targetinv - inventory ?x - object ?tool - object ?target - object ?t - tile)
  :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x IronOreVein)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Pickaxe)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target IronOre)
  )
 )
(:action mine-wood_2
 :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?targetinv - inventory ?target - object)

 :precondition (and
   (agent-at ?t)
   (object-at ?x ?t)
   (object-of-type ?x Tree)
   (inventory-holding ?toolinv ?tool)
   (object-of-type ?tool Axe)
   (inventory-empty ?targetinv)
   (object-of-type ?target Hypothetical)
 )
 :effect (and
   (not (inventory-empty ?targetinv))
   (inventory-holding ?targetinv ?target)
   (not (object-of-type ?target Hypothetical))
   (object-of-type ?target Wood)
 )
)
(:action mine-wool1_0
 :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?targetinv - inventory ?target - object)

 :precondition (and
   (agent-at ?t)
   (object-at ?x ?t)
   (object-of-type ?x Sheep)
   (inventory-holding ?toolinv ?tool)
   (object-of-type ?tool Shears)
   (inventory-empty ?targetinv)
   (object-of-type ?target Hypothetical)
 )
 :effect (and
   (not (inventory-empty ?targetinv))
   (inventory-holding ?targetinv ?target)
   (not (object-of-type ?target Hypothetical))
```

```
    (object-of-type ?target Wool)
  )
 )
 (:action mine-potato_0
  :parameters (?t - tile ?x - object ?targetinv - inventory ?target -
     object)

  :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x PotatoPlant)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Potato)
  )
 )
 (:action mine-sugar-cane_2
  :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
     targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x SugarCanePlant)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Axe)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target SugarCane)
  )
 )
 (:action mine-beetroot_1
  :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
     targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x BeetrootCrop)
    (inventory-holding ?toolinv ?tool)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Beetroot)
  )
 )
 (:action mine-feather_1
  :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
     targetinv - inventory ?target - object)

  :precondition (and
```

```
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x Chicken)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Sword)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Feather)
  )
)
(:action mine-cobblestone_2
 :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
    targetinv - inventory ?target - object)

 :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x CobblestoneStash)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Pickaxe)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Cobblestone)
  )
)
(:action mine-gold-ore1_2
 :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
    targetinv - inventory ?target - object)

 :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x GoldOreVein)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Pickaxe)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target GoldOre)
  )
)
(:action mine-coal1_0
 :parameters (?t - tile ?x - object ?toolinv - inventory ?tool - object ?
    targetinv - inventory ?target - object)

 :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x CoalOreVein)
    (inventory-holding ?toolinv ?tool)
    (object-of-type ?tool Pickaxe)
```

```
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Coal)
  )
)
(:action mine-beetroot1_0
 :parameters (?t - tile ?x - object ?targetinv - inventory ?target -
    object)

 :precondition (and
    (agent-at ?t)
    (object-at ?x ?t)
    (object-of-type ?x BeetrootCrop)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Beetroot)
  )
)
(:action craft-wood-plank
  :parameters (?ingredientinv1 - inventory ?targetinv - inventory ?
    station - object ?ingredient1 - object ?target - object ?t - tile)
  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 Wood)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target WoodPlank)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 Wood))
    (object-of-type ?ingredient1 Hypothetical)
  )
 )
(:action craft-arrow
  :parameters (?ingredientinv1 - inventory ?ingredientinv2 - inventory ?
    targetinv - inventory ?station - object ?ingredient1 - object ?
    ingredient2 - object ?target - object ?t - tile)
  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 Stick)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 Feather)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
```

```
    )
    :effect (and
      (not (inventory-empty ?targetinv))
      (inventory-holding ?targetinv ?target)
      (not (object-of-type ?target Hypothetical))
      (object-of-type ?target Arrow)
      (not (inventory-holding ?ingredientinv1 ?ingredient1))
      (inventory-empty ?ingredientinv1)
      (not (object-of-type ?ingredient1 Stick))
      (object-of-type ?ingredient1 Hypothetical)
      (not (inventory-holding ?ingredientinv2 ?ingredient2))
      (inventory-empty ?ingredientinv2)
      (not (object-of-type ?ingredient2 Feather))
      (object-of-type ?ingredient2 Hypothetical)
    )
  )
  (:action craft-beetroot-soup_0
   :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
      ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
      object ?targetinv - inventory ?target - object)

   :precondition (and
     (agent-at ?t)
     (object-at ?station ?t)
     (object-of-type ?station WorkStation)
     (inventory-holding ?ingredientinv1 ?ingredient1)
     (object-of-type ?ingredient1 Beetroot)
     (inventory-holding ?ingredientinv2 ?ingredient2)
     (object-of-type ?ingredient2 Bowl)
     (inventory-empty ?targetinv)
     (object-of-type ?target Hypothetical)
   )
   :effect (and
     (not (inventory-empty ?targetinv))
     (inventory-holding ?targetinv ?target)
     (not (object-of-type ?target Hypothetical))
     (object-of-type ?target BeetrootSoup)
     (not (inventory-holding ?ingredientinv1 ?ingredient1))
     (inventory-empty ?ingredientinv1)
     (not (object-of-type ?ingredient1 Beetroot))
     (object-of-type ?ingredient1 Hypothetical)
     (not (inventory-holding ?ingredientinv2 ?ingredient2))
     (inventory-empty ?ingredientinv2)
     (not (object-of-type ?ingredient2 Bowl))
     (object-of-type ?ingredient2 Hypothetical)
   )
  )
  (:action craft-paper_0
   :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
      ingredient1 - object ?targetinv - inventory ?target - object)

   :precondition (and
     (agent-at ?t)
     (object-at ?station ?t)
     (object-of-type ?station WorkStation)
     (inventory-holding ?ingredientinv1 ?ingredient1)
     (object-of-type ?ingredient1 SugarCane)
     (inventory-empty ?targetinv)
     (object-of-type ?target Hypothetical)
   )
   :effect (and
     (not (inventory-empty ?targetinv))
     (inventory-holding ?targetinv ?target)
     (not (object-of-type ?target Hypothetical))
     (object-of-type ?target Paper)
```

```
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 SugarCane))
    (object-of-type ?ingredient1 Hypothetical)
  )
 )
 (:action craft-shears2_2
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 GoldIngot)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Shears)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 GoldIngot))
    (object-of-type ?ingredient1 Hypothetical)
  )
 )
 (:action craft-bowl_1
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 WoodPlank)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 WoodPlank)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Bowl)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 WoodPlank))
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
    (not (object-of-type ?ingredient2 WoodPlank))
    (object-of-type ?ingredient2 Hypothetical)
  )
 )
 (:action craft-boat_0
  :parameters (?t - tile ?station - object ?ingredientinv - inventory ?
     ingredient - object ?targetinv - inventory ?target - object)

```

```
  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv ?ingredient)
    (object-of-type ?ingredient WoodPlank)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Boat)
    (not (inventory-holding ?ingredientinv ?ingredient))
    (inventory-empty ?ingredientinv)
    (not (object-of-type ?ingredient WoodPlank))
    (object-of-type ?ingredient Hypothetical)
  )
)
(:action craft-cooked-potato_1
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 Potato)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 Coal)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target CookedPotato)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 Potato))
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
    (not (object-of-type ?ingredient2 Coal))
    (object-of-type ?ingredient2 Hypothetical)
  )
)
(:action craft-gold-ingot_1
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 GoldOre)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 Coal)
    (inventory-empty ?targetinv)
```

```
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target GoldIngot)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 GoldOre))
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
    (not (object-of-type ?ingredient2 Coal))
    (object-of-type ?ingredient2 Hypothetical)
  )
)
(:action craft-stick_0
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 WoodPlank)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Stick)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 WoodPlank))
    (object-of-type ?ingredient1 Hypothetical)
  )
)
(:action craft-sword_0
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 Stick)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 IronIngot)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Sword)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 Stick))
```

```
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
    (not (object-of-type ?ingredient2 IronIngot))
    (object-of-type ?ingredient2 Hypothetical)
  )
 )
 (:action craft-bed_1
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 WoodPlank)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 Wool)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target Bed)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 WoodPlank))
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
    (not (object-of-type ?ingredient2 Wool))
    (object-of-type ?ingredient2 Hypothetical)
  )
 )
 (:action craft-iron-ingot_2
  :parameters (?t - tile ?station - object ?ingredientinv1 - inventory ?
     ingredient1 - object ?ingredientinv2 - inventory ?ingredient2 -
     object ?targetinv - inventory ?target - object)

  :precondition (and
    (agent-at ?t)
    (object-at ?station ?t)
    (object-of-type ?station WorkStation)
    (inventory-holding ?ingredientinv1 ?ingredient1)
    (object-of-type ?ingredient1 IronOre)
    (inventory-holding ?ingredientinv2 ?ingredient2)
    (object-of-type ?ingredient2 Coal)
    (inventory-empty ?targetinv)
    (object-of-type ?target Hypothetical)
  )
  :effect (and
    (not (inventory-empty ?targetinv))
    (inventory-holding ?targetinv ?target)
    (not (object-of-type ?target Hypothetical))
    (object-of-type ?target IronIngot)
    (not (inventory-holding ?ingredientinv1 ?ingredient1))
    (inventory-empty ?ingredientinv1)
    (not (object-of-type ?ingredient1 IronOre))
    (object-of-type ?ingredient1 Hypothetical)
    (not (inventory-holding ?ingredientinv2 ?ingredient2))
    (inventory-empty ?ingredientinv2)
```

```
    (not (object-of-type ?ingredient2 Coal))
    (object-of-type ?ingredient2 Hypothetical)
  )
)
)
```

Learned Operator Libraries on ALFRED The following shows the full PDDL domain definition including the initial provided vocabulary of symbolic environment constants and predicates, initial pick and place operators, and all ensuing learned operators.

```
(define (domain alfred)
    (:requirements :adl
    )
    (:types
        agent location receptacle object rtype otype
    )
    (:constants
        CandleType - otype
        ShowerGlassType - otype
        CDType - otype
        TomatoType - otype
        MirrorType - otype
        ScrubBrushType - otype
        MugType - otype
        ToasterType - otype
        PaintingType - otype
        CellPhoneType - otype
        LadleType - otype
        BreadType - otype
        PotType - otype
        BookType - otype
        TennisRacketType - otype
        ButterKnifeType - otype
        ShowerDoorType - otype
        KeyChainType - otype
        BaseballBatType - otype
        EggType - otype
        PenType - otype
        ForkType - otype
        VaseType - otype
        ClothType - otype
        WindowType - otype
        PencilType - otype
        StatueType - otype
        LightSwitchType - otype
        WatchType - otype
        SpatulaType - otype
        PaperTowelRollType - otype
        FloorLampType - otype
        KettleType - otype
        SoapBottleType - otype
        BootsType - otype
        TowelType - otype
        PillowType - otype
        AlarmClockType - otype
        PotatoType - otype
        ChairType - otype
        PlungerType - otype
        SprayBottleType - otype
        HandTowelType - otype
        BathtubType - otype
        RemoteControlType - otype
        PepperShakerType - otype
        PlateType - otype
```

```
        BasketBallType - otype
        DeskLampType - otype
        FootstoolType - otype
        GlassbottleType - otype
        PaperTowelType - otype
        CreditCardType - otype
        PanType - otype
        ToiletPaperType - otype
        SaltShakerType - otype
        PosterType - otype
        ToiletPaperRollType - otype
        LettuceType - otype
        WineBottleType - otype
        KnifeType - otype
        LaundryHamperLidType - otype
        SpoonType - otype
        TissueBoxType - otype
        BowlType - otype
        BoxType - otype
        SoapBarType - otype
        HousePlantType - otype
        NewspaperType - otype
        CupType - otype
        DishSpongeType - otype
        LaptopType - otype
        TelevisionType - otype
        StoveKnobType - otype
        CurtainsType - otype
        BlindsType - otype
        TeddyBearType - otype
        AppleType - otype
        WateringCanType - otype
        SinkType - otype

        ArmChairType - rtype
        BedType - rtype
        BathtubBasinType - rtype
        DresserType - rtype
        SafeType - rtype
        DiningTableType - rtype
        SofaType - rtype
        HandTowelHolderType - rtype
        StoveBurnerType - rtype
        CartType - rtype
        DeskType - rtype
        CoffeeMachineType - rtype
        MicrowaveType - rtype
        ToiletType - rtype
        CounterTopType - rtype
        GarbageCanType - rtype
        CoffeeTableType - rtype
        CabinetType - rtype
        SinkBasinType - rtype
        OttomanType - rtype
        ToiletPaperHangerType - rtype
        TowelHolderType - rtype
        FridgeType - rtype
        DrawerType - rtype
        SideTableType - rtype
        ShelfType - rtype
        LaundryHamperType - rtype

    )
    ;; Predicates defined on this domain. Note the types for each
    predicate.
```

```
(:predicates
    (atLocation ?a - agent ?l - location)
        (receptacleAtLocation ?r - receptacle ?l - location)
        (objectAtLocation ?o - object ?l - location)
        (inReceptacle ?o - object ?r - receptacle)
        (receptacleType ?r - receptacle ?t - rtype)
        (objectType ?o - object ?t - otype)
        (holds ?a - agent ?o - object)
        (holdsAny ?a - agent)
        (holdsAnyReceptacleObject ?a - agent)

        (openable ?r - receptacle)
        (opened ?r - receptacle)
        (isClean ?o - object)
        (cleanable ?o - object)
        (isHot ?o - object)
        (heatable ?o - object)
        (isCool ?o - object)
        (coolable ?o - object)
        (toggleable ?o - object)
        (isToggled ?o - object)
        (sliceable ?o - object)
        (isSliced ?o - object)
 )
(:action PickupObjectNotInReceptacle
        :parameters (?a - agent ?l - location ?o - object)
        :precondition (and
            (atLocation ?a ?l)
            (objectAtLocation ?o ?l)
            (not (holdsAny ?a))
            (forall
                (?re - receptacle)
                (not (inReceptacle ?o ?re))
            )
        )
        :effect (and
            (not (objectAtLocation ?o ?l))
            (holds ?a ?o)
            (holdsAny ?a)
        )
    )

(:action PutObjectInReceptacle
        :parameters (?a - agent ?l - location ?ot - otype ?o - object ?r
    - receptacle)
        :precondition (and
            (atLocation ?a ?l)
            (receptacleAtLocation ?r ?l)
            (objectType ?o ?ot)
            (holds ?a ?o)
            (not (holdsAnyReceptacleObject ?a))
        )
        :effect (and
            (inReceptacle ?o ?r)
            (not (holds ?a ?o))
            (not (holdsAny ?a))
            (objectAtLocation ?o ?l)
        )
    )

(:action PickupObjectInReceptacle
        :parameters (?a - agent ?l - location ?o - object ?r - receptacle
    )
        :precondition (and
            (atLocation ?a ?l)
```

```
            (objectAtLocation ?o ?l)
            (inReceptacle ?o ?r)
            (not (holdsAny ?a))
        )
        :effect (and
            (not (objectAtLocation ?o ?l))
            (not (inReceptacle ?o ?r))
            (holds ?a ?o)
            (holdsAny ?a)
        )
    )

    (:action RinseObject_2
        :parameters (?toolreceptacle - receptacle ?a - agent ?l - location ?o - object)

        :precondition (and
        (receptacleType ?toolreceptacle SinkBasinType)
        (atLocation ?a ?l)
        (receptacleAtLocation ?toolreceptacle ?l)
        (objectAtLocation ?o ?l)
        (cleanable ?o)
        )
        :effect (and
        (isClean ?o)
        )
)

(:action TurnOnObject_2
        :parameters (?a - agent ?l - location ?o - object)

        :precondition (and
        (atLocation ?a ?l)
        (objectAtLocation ?o ?l)
        (toggleable ?o)
        )
        :effect (and
        (isToggled ?o)
        )
)

(:action CoolObject_0
        :parameters (?toolreceptacle - receptacle ?a - agent ?l - location ?o - object)

        :precondition (and
        (receptacleType ?toolreceptacle FridgeType)
        (atLocation ?a ?l)
        (receptacleAtLocation ?toolreceptacle ?l)
        (holds ?a ?o)
        )
        :effect (and
        (isCool ?o)
        )
)
(:action SliceObject_1
        :parameters (?toolobject - object ?a - agent ?l - location ?o - object)

        :precondition (and
        (objectType ?toolobject ButterKnifeType)
        (atLocation ?a ?l)
        (objectAtLocation ?o ?l)
        (sliceable ?o)
        (holds ?a ?toolobject)
```

```
        )
        :effect (and
        (isSliced ?o)
        )
)
(:action SliceObject_0
        :parameters (?toolobject - object ?a - agent ?l - location ?o -
    object)

        :precondition (and
        (objectType ?toolobject KnifeType)
        (atLocation ?a ?l)
        (objectAtLocation ?o ?l)
        (sliceable ?o)
        (holds ?a ?toolobject)
        )
        :effect (and
        (isSliced ?o)
        )
)
(:action MicrowaveObject_0
        :parameters (?toolreceptacle - receptacle ?a - agent ?l -
    location ?o - object)

        :precondition (and
        (receptacleType ?toolreceptacle MicrowaveType)
        (atLocation ?a ?l)
        (receptacleAtLocation ?toolreceptacle ?l)
        (holds ?a ?o)
        )
        :effect (and
        (isHot ?o)
        )
)

)
```