Informing Reinforcement Learning Agents by Grounding Language to Markov Decision Processes

Benjamin A. Spiegel, Ziyi Yang, William Jurayj, Ben Bachmann,
Stefanie Tellex, George Konidaris
Department of Computer Science
Brown University

Abstract

While significant efforts have been made to leverage natural language to accelerate reinforcement learning, utilizing diverse forms of language efficiently remains unsolved. Existing methods focus on mapping natural language to individual elements of MDPs such as reward functions or policies, but such approaches limit the scope of language they consider to make such mappings possible. We present an approach for leveraging general language advice by translating sentences to a grounded formal language for expressing information about *every* element of an MDP and its solution including policies, plans, reward functions, and transition functions. We also introduce a new model-based reinforcement learning algorithm, RLang-Dyna-Q, capable of leveraging all such advice, and demonstrate in two sets of experiments that grounding language to every element of an MDP leads to significant performance gains.

1 Introduction

Language serves as a powerful means for humans to share information about the world. Our grasp of language allows us to learn more quickly or even skip learning altogether, performing new tasks with ease by drawing upon the domain expertise of others in the form of advice. An open question in reinforcement learning is how language advice can be leveraged to speed up learning in Markov Decision Processes (MDPs), as learning tasks *tabula rasa* is exceptionally difficult—and often impossible—in the real world. While many methods of leveraging advice for learning have emerged in the literature, a coherent theory of *language grounding* that can comprehensively support the use of language for reinforcement learning has not.

Virtually all research in language and RL grounds language to individual elements of MDPs such as policies (Liang et al., 2023; Vemprala et al., 2023; Wu et al., 2023; Andreas et al., 2017), reward functions (Squire et al., 2015), and goals (Colas et al., 2020). The main drawbacks of these works is that they restrict their approach to narrow fragments of natural language. For example, a statement like *"if a mug is tipped over, its contents will spill out"* clearly refers to a transition function, and mapping this information to a policy is not straightforward. For this reason, works that ground language to policies primarily focus on *imperative* sentences that naturally correspond to policies, plans, or reward functions. Likewise, works that ground language to transition functions focus mainly on *declarative* sentences, which may provide information about the dynamics of a domain. This divergence in methodology suggests that not all language should be grounded to the same component of an MDP, and that a general language grounding system for reinforcement learning agents should be capable of grounding language to *every* element of an MDP and its solution.

We propose a novel approach to grounding natural language for use in reinforcement learning that formulates the language grounding problem as a machine translation task from natural language to RLang (Rodriguez-Sanchez et al., 2023), a formal language designed to express information about every element of an MDP and its solution. Our approach is akin to semantic parsing (Mooney,

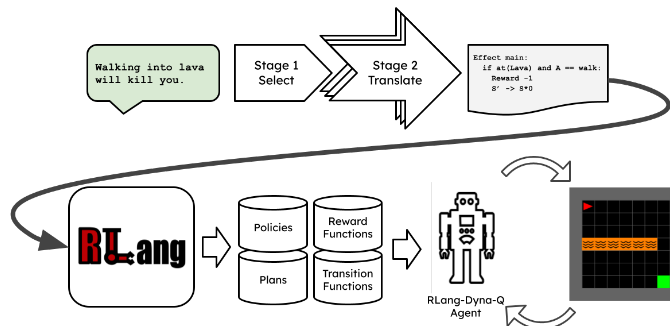

Figure 1: Our pipeline for translating natural language advice to RLang. We extend the original RLang pipeline to include natural language translation and a Dyna-Q agent capable of leveraging all forms of RLang advice.

2007)—a problem in natural language understanding that involves translating natural language into a formal representation—because RLang is a grounded formal language that offers a systematic means of expressing knowledge about an MDP. Such an approach calls for a learning agent capable of leveraging all such MDP components, including a partial policy, reward function, plan, and transition function. We therefore also introduce RLang-Dyna-Q, a model-based tabular RL agent based on Dyna-Q Sutton et al. (1998), that can effectively leverage such advice. We demonstrate the strength and generality of our approach by grounding a variety of natural language advice to RLang programs, which RLang-Dyna-Q can use to significantly improve performance, sometimes making it possible to solve tasks that vanilla Dyna-Q cannot solve.

2 Background

Reinforcement learning tasks are typically modeled as Markov decision processes (MDPs), which can be represented by a tuple $\langle S, A, R, T, \gamma \rangle$, where S is the set of states, A is the set of actions, R is the reward function, T is the transition function, and γ is the discount factor. The goal of an agent is to find a policy, $\pi(a|s)$—a function that selects an action for each state—which maximizes the expected sum of discounted rewards:

$$\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \right].$$

Value-based reinforcement learning algorithms rely on estimating the optimal action-value function q_*, defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a),$$

providing the expected return for taking action a in state s and subsequently following an optimal policy (Sutton et al., 1998). Q-learning (Watkins, 1989) works to approximate q_* by applying the following update rule after taking roll-outs in the environment:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \big].$$

Building on Q-learning, Dyna-Q (Sutton et al., 1998) introduces an additional component: a model of the environment. While Q-learning learns from direct interaction with the environment alone, Dyna-Q builds an internal model of the environment and updates the action-value function using both real and simulated roll-outs, enabling faster convergence to the optimal action-value function.

2.1 Large Language Models for Machine Translation

Large Language Models (LLMs), often based on architectures like the Transformer (Vaswani et al., 2017), are trained to predict the next token x_t in a sequence given the preceding tokens

Table 1: Selected MDP elements, corresponding RLang groundings, and natural language interpretations. The first column shows a component of the MDP, the second shows an RLang expression that can inform it, and the last column contains a description of the expression.

MDP Component	**RLang Declaration**	**Natural Language Interpretation**

Policy
$\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$
	`Policy build_bridge:`
	`  if at_workbench:`
	`    Execute use`
	`  else:`
	`    Execute go_to(workbench)`
		If you are at a workbench, use it. Otherwise, go to it.

Plan
$\{A_0, A_1, ..., A_n\}$
	`Plan gather_materials:`
	`  Execute go_to(wood)`
	`  Execute pickup`
	`  Execute go_to(string)`
	`  Execute pickup`
		Go to the wood and pick it up, then go to the string and pick it up.

Reward, Transition Func.
$R_e : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow$
$T_e : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$
	`Effect common_sense:`
	`  if at(Wall) and A == walk:`
	`    Reward 0, S' -> S`
	`  if at(Lava) and A == walk:`
	`    Reward -1, S' -> S*0`
		Walking into walls will get you nowhere. Walking into lava will kill you.

$\{x_1, x_2, ..., x_{t-1}\}$ via the following objective:

$$\mathcal{L} = -\sum_t \log P(x_t | x_1, x_2, \ldots, x_{t-1}).$$

In very large models, this objective results in emergent capabilities such as natural language understanding and generation, making them suitable for a variety of tasks beyond mere text completion including question-answering, summarization, and more (Bubeck et al., 2023). One useful emergent capability of LLMs is the machine translation of text from one language to another. While specialized neural machine translation systems are trained using a parallel corpus to maximize the conditional probability $P(y|x)$, where x is the source sequence and y is the target sequence (Bahdanau et al., 2014), LLMs have achieved similar translation capabilities despite not being trained explicitly on this objective (Brown et al., 2020). Furthermore LLMs have been shown to be proficient at generating text in formal languages such as Python given a language prompt (Chen et al., 2021; Li et al., 2023).

2.2 Leveraging Formal Specification Languages for Decision-Making

Formal specification languages have long been a useful tool to inform decision-making agents. In classical planning, for example, it is standard to use the Planning Domain Description Language (PDDL; Ghallab et al. 1998) and its probabilistic extension PPDDL (probabilistic PDDL; Younes & Littman, 2004) to specify the complete dynamics of an environment. Other languages like Linear Temporal Logic (LTL; Littman et al., 2017; Jothimurugan et al., 2019) and Policy Sketches (Andreas et al., 2017) are sufficient for describing goals and hierarchical policies, respectively, for instruction-following agents. While effective, one limiting factor of these formal languages is their narrow scope. Natural language, by contrast, can be used to express information about nearly *all* the elements of decision-making.

RLang (Rodriguez-Sanchez et al., 2023) is a recent formal language to emerge from the literature. While previous languages for decision-making narrowly focus on individual components of an MDP such as a policy or reward function, RLang was designed to provide information about *every* component of a structured MDP and its solution. Formally, an RLang specification is a set of RLang groundings $\mathcal{G}$ given by an RLang program $\mathcal{P}$ and an RLang vocabulary $\mathcal{V}$, which may include additional groundings for use across multiple MDPs. Some example RLang programs and their natural language interpretations can be seen in Table 1. Crucially, advice specified by RLang can

be compiled directly into many components of an MDP including policies, transition functions, reward functions, and plans. Leveraging such components in a learning algorithm is not always straightforward, however, and integrating more than one component into an agent is a non-trivial problem that the original authors did not solve.

3 Grounding Natural Language Advice to RLang Programs

One major motivation for leveraging language advice in reinforcement learning is to supply agents with the kinds of commonsense reasoning that language can easily express. Consider the LavaCrossing environment in Figure 6. Any human interacting with this environment would quickly learn that walking into the lava squares kills you, or likewise that walking into walls will do nothing at all. Communicating this knowledge to others with language is natural for humans, but leveraging such language advice in reinforcement learning is a major unsolved problem. An alternative approach to supplying commonsense advice to RL agents involves specifying it in a formal language relevant to decision-processes, which can more straightforwardly be used by a learning agent to improve learning. Such an approach is limited only by the expressivity of the formal language and how it is used by the learning agent.

As formal languages for decision-making grow more expressive, a natural next step for leveraging language advice in reinforcement learning is to translate pieces of natural language advice into statements in such formal languages. RLang is a highly expressive candidate for language grounding because it is capable of specifying information about *every* element of a structured MDP and its solution, including plans, policies, transition functions, and reward functions (see Table 2 in Rodriguez-Sanchez et al. (2023)). Furthermore, we hypothesize that different kinds of advice can most naturally be represented by different components of the MDP, and that methods that ground language to a single component are insufficient to capture general language advice. For example, the statement, “stacked dishes can topple if unevenly piled," is precisely a statement about transition dynamics, and while it can be used ultimately in a plan or policy, the information contained in the statement would not be retrievable if it were not represented as a partial transition function. Likewise, the sentence, “wear oven mitts whenever handling pots and pans," is a statement about a policy, and representing it as a reward function would only indirectly capture this advice.

We therefore formulate the language grounding problem in RL as a machine translation task from natural language to RLang. Our task is as follows: given an RLang vocabulary $\mathcal{V}$—a set of task-general groundings that act as primitives in an RLang program—for a given MDP and a piece of natural language advice u, we seek a function $\phi : u \times \mathcal{V} \to \mathcal{P}_u$, where $\mathcal{P}_u$ is an executable RLang program capturing the advice in u that can be leveraged by a learning agent. We propose to do this translation using a pre-trained large language model in a two-stage pipeline by 1) identifying which RLang grounding type would best capture the language advice; and 2) few-shot translating the advice into an RLang program. Stage 1, the selection stage, instructs the LLM to classify a novel piece of advice u into RLang grounding types such as Effects, Policies, and Plans, consulting a small number of example classifications in the prompt. This ensures that the advice will be represented by an appropriate component of the MDP.[1] Stage 2, the translation stage, instructs the LLM to translate u to an RLang program specifying the grounding type given by Stage 1 using roughly 5 example translations in the prompt that were hand-engineered to cover a wide range of RLang's syntax. In experiments we demonstrate that this pipeline effectively grounds the advice, yielding an RLang program that may contain partial transition functions, reward functions, policies, and plans. Our pipeline is illustrated in Figure 1.

3.1 RLang-Dyna-Q: A Single Agent for Leveraging All of RLang

In the original RLang paper, the authors presented a number of RLang-enabled agents—including ones based on Q-Learning, PPO (Schulman et al., 2017), and DOORmax (Diuk et al., 2008)—each

[1] We assume that each piece of advice—which may contain multiple sentences—grounds to a single RLang grounding type. This constraint can easily be relaxed in future work.

capable of leveraging *individual* RLang groundings to improve learning. However, leveraging general language advice requires integrating potentially *all* RLang groundings into a single learning agent. We therefore introduce RLang-Dyna-Q, a learning agent based on Dyna-Q (Sutton et al., 1998) that is capable of simultaneously leveraging a partial policy, plan, reward function, and transition function given by an RLang program. Dyna-Q is an appropriate core learning agent because integrating actions and dynamics is most natural in a model-based learning algorithm that explicitly represents a policy, transition function, and reward function (see Algorithm 1, our modifications to Dyna-Q are in blue).

4 Experiments

We hypothesize that RLang is an effective grounding for natural language advice in the context of reinforcement learning. However, evaluating whether language advice u and RLang program $\mathcal{P}_u$ have the same semantic content is difficult, so we designed our experiments to test a proxy objective of primary interest: the agent's performance on a learning task. If we provide advice that is helpful to the agent, then grounding it properly should improve performance. We therefore assessed our translation pipeline by evaluating agent performance on multiple custom tasks based on the Minigrid/BabyAI (Chevalier-Boisvert et al., 2023; 2018) and VirtualHome (Puig et al., 2018) environments given expert advice. We include ablations of different RLang components to demonstrate our auxiliary hypothesis, that language is best grounded to every element of an MDP. We also run a small user study to assess our pipeline's efficacy on a wide range of non-expert language advice.

4.1 Leveraging Expert Advice in Minigrid

We designed custom environments using the Minigrid/BabyAI library, a platform for studying the behavior of language-informed agents. In a typical Minigrid environment, an agent might reason about opening and closing doors using keys which may be hidden in other rooms, managing a small inventory of items, removing obstacles like balls out of the way to reach other rooms or objects, and avoiding lava, all for the ultimate purpose of reaching a goal. Minigrid environments are an ideal setting for our experiments for three reasons: 1) they can be solved using tabular RL algorithms, which our informed, model-based RLang-Dyna-Q agent is based on; 2) there are clear and obvious referents of language in both the state and action spaces of these environments (e.g. keys, doors, and balls are represented neatly in a discrete state space and skills such as walking towards objects are easy to implement); 3) many objects are shared across environments enabling the reuse of a common RLang vocabulary for referencing these objects, which makes it easier for our translation pipeline to ground novel advice.

For each environment, we provide an RLang vocabulary file $\mathcal{V}$, a set of RLang groundings to be used as primitives in a full RLang program. These vocabulary files are generated automatically for each minigrid environment given a single general template, and include perception abstractions such as the objects in the environment (e.g., `yellow_key`, `red_door`) and a short list of predicates for reasoning with them (e.g., `carrying()`, `reachable()`, `at()`), as well as a single abstract action in the form of a lifted skill for walking to any reachable object (`go_to()`). Importantly, these groundings have semantically-meaningful labels, which enable a simple translation process. In our final experiment, we relax this constraint somewhat, by determining the semantic label of some groundings with an off-the-shelf vision model. All agents in the experiments, including the Random, Dyna-Q, and RLang-Dyna-Q agents, are given access to this lifted skill. However, we do not provide the Dyna-Q agent with any perception abstractions, as including them induces an equivalent state space in the tabular RL setting. Likewise, the Random agent does not consider state when selecting an action. In Stage 2 of the translation pipeline, we provide the list of available RLang groundings that can be referenced in an RLang program along with the language advice. This prevents the LLM from hallucinating imaginary skills, objects, and predicates when translating the advice into an RLang program. The LLM never interacts with the MDP directly. The translation examples used in the prompts in both stages of translation did not change across experiments, though these

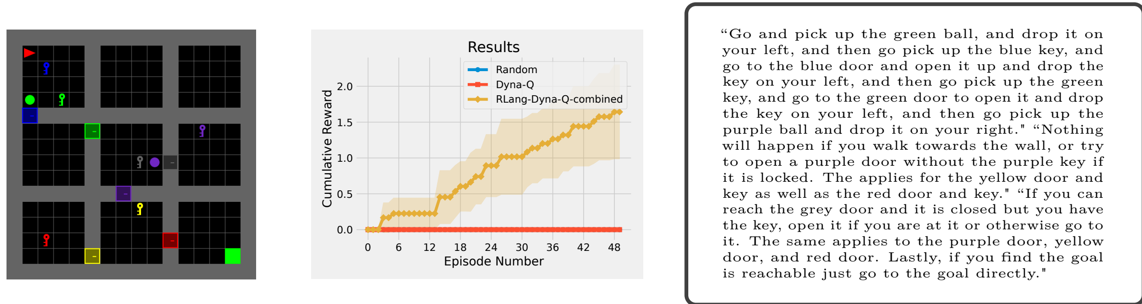

Figure 2: **HardMaze Experiment.** Language advice given to the agent was grounded to RLang effects, plans, and policies. The full translated RLang program is available in the appendix. Vanilla Dyna-Q was not able to complete this task. The initial state of HardMaze is pictured on the left, reward curves are in the center, and the language advice given is on the right.

translations are vocabulary-specific and grounding advice to environments outside of Minigrid will require domain-compatible example translations.

We evaluated our grounding pipeline on four diverse Minigrid environments: LavaCrossing, MultiRoom, MidMazeLava, and HardMaze. For each environment, we collected multiple pieces of natural language advice from human experts and translated them into RLang programs using our two-stage pipeline. Each piece of advice was translated to a single RLang grounding type, and each piece of advice contained multiple sentences. We then evaluated our RLang-Dyna-Q agent on each environment with the translated RLang programs. In the MidMazeLave environment, advice was grounded to multiple RLang types. We include additional results for RLang-Dyna-Q utilizing only one type of advice at a time—Effects, Plans, or Policies—to isolate the impact of each on performance.

In all of the experiments, RLang-Dyna-Q significantly outperformed vanilla Dyna-Q. In LavaCrossing (see Figure 6), the agent is tasked with reaching a goal while avoiding lava, and merely advising the agent about the dangers of lava and futility of walking into walls greatly increases performance. In the MultiRoom environment (see Figure 8), in which the agent must open a series of doors to reach a goal, providing a plan in natural language significantly increased performance. In MidMazeLava (see Figure 7) and HardMaze (see Figure 2), the agent is faced with significantly more difficult tasks. In the former, the agent must unblock doors and open them with keys to reach a goal while avoiding lava, and in the latter the agent must traverse through many rooms, bringing keys across rooms to doors which must be unblocked to reach a goal. We collected paragraphs-worth of advice for these environments, which we translated into RLang plans, policies, and effects. In HardMaze, this language advice made it possible to solve the task, as the vanilla Dyna-Q agent did no better than random. For each experiment, 10 instances of each agent were run to generate a 95% confidence interval on their cumulative reward over 50 episodes (LavaCrossing was run for 25 episodes only). The number of timesteps per episode varied across environments.

4.2 Leveraging Expert Advice in VirtualHome

We ran additional experiments on custom environments based on the VirtualHome library, a platform for simulating complex household activities. VirtualHome has an object-oriented state space, which can be referenced natively in RLang. We engineered 2 tasks in a kitchen environment to assess our language grounding pipeline: FoodSafety (see Figure 3), where the agent is tasked with putting a pie into the fridge and salmon in to the microwave, and CouchPotato, where the agent is tasked with bringing a remote control to a sofa and putting cereal into a kitchen cabinet, while avoiding picking up toothpaste. In these environments, agents are given an RLang vocabulary file with groundings for object-oriented perception and action abstractions such as the objects in the environment (e.g. `salmon_327`, `fridge_305`), a short list of predicates (e.g. `inside()`, `holding()`, `near()`), and a set

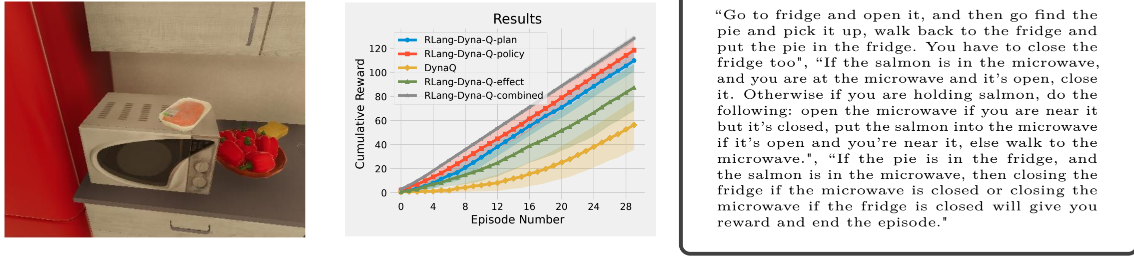

Figure 3: **FoodSafety Experiment.** Language advice given to the agent was grounded to RLang effects, plans, and policies. The full translated RLang program is in the appendix. All RLang-Dyna-Q agents outperformed Dyna-Q. The FoodSafety environment is pictured on the left, reward curves are center, and the language advice given is on the right.

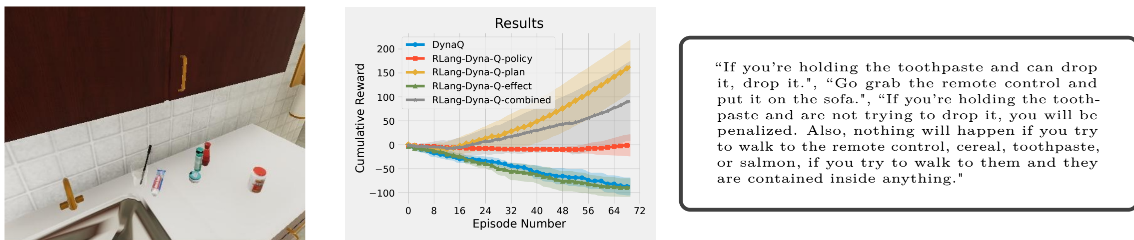

Figure 4: **CouchPotato Experiment.** Language advice given to the agent was grounded to RLang effects, plans, and policies. All the RLang agents outperformed Dyna-Q with the exception of the Effect-enabled agent. We note that bugs in the simulator non-deterministically prevent certain actions from executing, so the advice we specify only applies part of the time.

of lifted skills (e.g. `walk_to()`, `open`, `grab`). These groundings have semantically meaningful labels, which make them easy targets for grounding natural language.

Our experimental design here is identical to the Minigrid experiments: for each environment we collected multiple pieces of language advice from human experts and translated them into RLang programs via our two-stage pipeline. We then evaluated the performance of an RLang-Dyna-Q agent on our environments in comparison to a vanilla Dyna-Q agent. In all of our experiments, the RLang-informed agents significantly outperformed Dyna-Q. For each experiment, 10 instances of each agent were run to generate a 95% confidence interval on their cumulative reward over 50 and 70 episodes for FoodSafety and CouchPotato, respectively. The maximum number of timesteps per episode varied. The agent parameters are listed in the appendix.

4.3 Grounding Symbols with a Vision-Language Model

A crucial assumption made by our pipeline is that we are given semantically-meaningful labels for the groundings we have, including labels for objects, skills, and predicates. Assigning relevant labels for these groundings enables a relatively simple translation from natural language into RLang. In a real-world setting, we imagine that the labels for these groundings can be generated in two ways: 1) prescriptively, in the case of skill engineering by humans, and 2) via a pre-trained foundation model for identifying predicates and objects in the environment. Implementing a full symbol-grounding system[2] is outside the scope of this work, however, we performed an additional experiment to demonstrate how the labels for object groundings could be easily extracted from images of the VirtualHome environments using a vision-language model. Details can be found in Appendix C.

[2]Learning a mapping from symbolic labels to groundings is explored in Steels & Hild (2012).

4.4 Evaluating Translation and RLang Efficacy

To assess RLang's ability to capture the breadth of general language advice, we ran a small user study. After explaining the controls for navigation, opening and closing doors, and picking up and dropping objects, we asked 10 undergraduate students to solve the LockedRoom MiniGrid task (pictured in Figure 5). We then asked them to describe in one or two sentences any advice they would give to an agent completing the task for the first time. We collected their responses and ran them through our translation pipeline to arrive at the RLang groundings in Table 3 of the Appendix. Of 10 pieces of advice collected, 9 were translated into valid RLang programs, while 1 referenced groundings that did not exist (e.g. `second_left_door`). We used the remaining valid RLang programs to inform 9 separate RLang-Dyna-Q agents that we compared against a baseline Dyna-Q agent given no advice. With a few exceptions, providing advice either did not meaningfully impact performance over the baseline or led to dramatic improvements in performance (see Table 2). In the cases where advice did not impact performance, advice was translated into a valid RLang program, but the program was missing a crucial step because it did not have an appropriate RLang grounding to reference (e.g. it had no way of translating "the room with the red key"). This is not a limitation of our pipeline, however, as a more expressive RLang vocabulary file could be used to achieve such a translation (e.g. `room_with(red_key)` can evaluate to `grey_door`).

5 Related Works

Language in RL Luketina et al. (2019) identify two variations of language usage in the reinforcement learning literature. The first, **language-conditional** RL, is one in which language use is a necessary component of the task. This includes environments where agents must execute commands in natural language (Mirchandani et al., 2021), or otherwise deal with language that is part of the MDP, e.g., in the observation or action space (Fulda et al., 2017; Kostka et al., 2017). The second variation is **language-assisted** RL, in which natural language is used to communicate task-related information to an agent that is *not necessary* for solving the task. In these settings, language can be used to inform policy structure (Watkins et al., 2023), reward functions (Goyal et al., 2019), transition dynamics (Narasimhan et al., 2018), or Q-functions (Branavan et al., 2012).

Grounding Natural to Formal Languages for Planning and Learning The notion of grounding natural language to a formal language for use in learning and planning is not new. Gopalan et al. (2018) and Berg et al. (2020) translate natural language commands into Linear Temporal Logic (LTL), which they use as reward functions for a learning agent or planning objectives, and Silver et al. (2023) and Miglani & Yorke-Smith (2020) ground natural language into PDDL, which is fed to a recurrent neural network to output solution plans. However, the advancement of large language models (LLMs) has led to even more capable agents that for leveraging formal languages. In the planning literature, Ahn et al. (2022); Huang et al. (2022); Song et al. (2023) use primitive formal languages for executing policies on real robots or in embodied environments, Liu et al. (2023a); Xie et al. (2023) translate natural language commands into PDDL plans with the help of LLMs, and Liu et al. (2023b) proposed a modular system to ground natural language into LTL formulas. Code is also a popular choice for formal languages: Liang et al. (2023); Vemprala et al. (2023); Wu et al. (2023) use an LLM to generate Python functions as policies from natural language instructions; Singh et al. (2022) also generates programs by prompting LLMs for code completion. For learning, more recent works focus on reward design with LLMs for RL agents: Yu et al. (2023) specifies reward with LLMs through code generation and Du et al. (2023) leverage commonsense reasoning for designing reward functions. While many methods excel at grounding to formal languages, no existing method seeks to ground language to every component of the MDP.

6 Discussion and Conclusion

Natural language grounding (Steels & Hild, 2012) has critical implications for all of AI. Just as RL is intended as a model of intelligent decision making, we propose that its core formalisms offer a

natural target for language grounding. If MDPs model human decision-making, and humans invented language to share information that aids their decision-making, then the appropriate target for language grounding should be an MDP, or a richer and perhaps more structured decision process reflecting the complexity of human decision-making. One line of evidence for this claim is the direct correspondence between parts of speech and elements of structured decision-processes (Rodriguez-Sanchez et al., 2020). For example, the object classes in Object Oriented MDPs (Diuk et al., 2008) naturally correspond to the concept of **common nouns** requiring **determiners** to single out class instances, and the parameters in Parameterized Action MDPs Masson et al. (2016) naturally correspond to **adverbs** for modifying the execution of discrete macro-actions (**verbs**).

More practically, knowledge expressed in natural language has immense potential to inform reinforcement learning agents, and thereby alleviate the high sample complexity of having to learn *tabula rasa*. We present a novel method for leveraging general natural language advice to expedite learning in Markov Decision Processes by translating it into RLang, a formal language designed to specify information about every element of an MDP and its solution. Our method can ground advice to reward functions, transition functions, plans, and policies. We also introduce a modified Dyna-Q agent capable of leveraging all of the types of information present in the partial MDP specification output by RLang. Our findings show that our approach can leverage a wide variety of language advice to accelerate learning.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 166–175. JMLR. org, 2017.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Matthew Berg, Deniz Bayazit, Rebecca Mathew, Ariel Rotter-Aboyoun, Ellie Pavlick, and Stefanie Tellex. Grounding Language to Landmarks in Arbitrary Outdoor Environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

S.R.K. Branavan, D. Silver, and R. Barzilay. Learning to win by reading manuals in a monte-carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704, apr 2012. doi: 10.1613/jair.3484. URL https://doi.org/10.1613%2Fjair.3484.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

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

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: A platform to study the sample efficiency of grounded language learning. *arXiv preprint arXiv:1810.08272*, 2018.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Cédric Colas, Ahmed Akakzia, Pierre-Yves Oudeyer, Mohamed Chetouani, and Olivier Sigaud. Language-conditioned goal generation: a new approach to language grounding for rl. *arXiv preprint arXiv:2006.07043*, 2020.

Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 240–247, 2008.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models, 2023.

Nancy Fulda, Daniel Ricks, Ben Murdoch, and David Wingate. What can you do with a rock? affordance extraction via word embeddings, 2017.

M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212.

Nakul Gopalan, Dilip Arumugam, Lawson Wong, and Stefanie Tellex. Sequence-to-Sequence Language Grounding of Non-Markovian Task Specifications. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, 2018. doi: 10.15607/RSS.2018.XIV.067.

Prasoon Goyal, Scott Niekum, and Raymond J. Mooney. Using natural language for reward shaping in reinforcement learning, 2019.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022.

Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Bartosz Kostka, Jaroslaw Kwiecieli, Jakub Kowalski, and Pawel Rychlikowski. Text-based adventures of the golovin AI agent. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, aug 2017. doi: 10.1109/cig.2017.8080433. URL https://doi.org/10.1109%2Fcig.2017.8080433.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023.

Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via gltl. *arXiv preprint arXiv:1704.04341*, 2017.

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, 2023a.

Jason Xinyu Liu, Ziyi Yang, Ifrah Idrees, Sam Liang, Benjamin Schornstein, Stefanie Tellex, and Ankit Shah. Lang2ltl: Translating natural language commands to temporal robot task specification, 2023b.

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 *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 6309–6317. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/880. URL https://doi.org/10.24963/ijcai.2019/880.

Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), Feb. 2016. doi: 10.1609/aaai.v30i1.10226. URL https://ojs.aaai.org/index.php/AAAI/article/view/10226.

Shivam Miglani and N. Yorke-Smith. Nltopddl: One-shot learning of pddl models from natural language process manuals. 2020. URL https://api.semanticscholar.org/CorpusID:231626460.

Suvir Mirchandani, Siddharth Karamcheti, and Dorsa Sadigh. ELLA: Exploration through learned language abstraction. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=VvUldGZ3izR.

Raymond J Mooney. Learning for semantic parsing. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pp. 311–324. Springer, 2007.

Karthik Narasimhan, Regina Barzilay, and Tommi Jaakkola. Grounding language for transfer in deep reinforcement learning, 2018.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8494–8502, 2018.

R. Rodriguez-Sanchez, B.A. Spiegel, J. Wang, R. Patel, G.D. Konidaris, and S. Tellex. Rlang: A declarative language for describing partial world knowledge to reinforcement learning agents. In *Proceedings of the Fortieth International Conference on Machine Learning*, July 2023.

Rafael Rodriguez-Sanchez, Roma Patel, and George Konidaris. On the relationship between structure in natural language and models of sequential decision processes. In *Language in Reinforcement Learning Workshop at ICML 2020*, 2020. URL https://openreview.net/forum?id=-KDnP4X1-0_.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

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, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models, 2022.

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, 2023.

Shawn Squire, Stefanie Tellex, Dilip Arumugam, and Lei Yang. Grounding english commands to reward functions. In *Robotics: Science and Systems*, 2015.

Luc Steels and Manfred Hild. *Language grounding in robots*. Springer Science & Business Media, 2012.

Richard S Sutton, Andrew G Barto, et al. *Introduction to Reinforcement Learning*, volume 135. MIT press Cambridge, 1998.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities, 2023.

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

Olivia Watkins, Trevor Darrell, Pieter Abbeel, Jacob Andreas, and Abhishek Gupta. Teachable reinforcement learning via advice distillation, 2023.

Jimmy Wu, Rika Antonova, Adam Kan, Marion Lepert, Andy Zeng, Shuran Song, Jeannette Bohg, Szymon Rusinkiewicz, and Thomas Funkhouser. Tidybot: Personalized robot assistance with large language models. *Autonomous Robots*, 2023.

Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models, 2023.

Håkan L. S. Younes and Michael L. Littman. Ppddl 1 . 0 : An extension to pddl for expressing planning domains with probabilistic effects. In *PPDDL 1 . 0 : An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects*, 2004.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis, 2023.

Table 2: **User Study.** We collected 10 pieces of advice from 10 undergraduate students for the LockedRoom environment. For each piece of advice, 5 agent instances were run for 25 episodes on the LockedRoom environment for 500 steps. The cumulative discounted reward for the 25 episodes is in the first column along with a 95% confidence interval. The average percent increase in cumulative discounted reward over the baseline is present in the second column. The second-to-last piece of advice did not ground to a valid RLang program, so no experiment was run.

Avg Cumulative Return	% improvement	**Natural Language Advice**
17.86 ± 2.36	—	No advice
22.01 ± 0.71	$+23.24$	"Remember to toggle to open doors."
16.79 ± 1.75	-5.99	"You don't need to carry keys to open the grey door."
17.22 ± 1.75	-3.60	"Identify the room with the red key, move to that room by opening the door. Pick up the key. Identify the room with the red door, proceed there. Open the red door. Find the green square and go there to finish the game."
23.55 ± 0.69	$+31.86$	"Move to the grey door, open it and enter the room until you get to the red key, pick it up. Exit the room and move towards the red door, open it and get into that room. Move to the green block and enter it."
23.93 ± 0.37	$+33.99$	"Go to the grey door. open the grey door. go to the red key. pick up the red key. go to the red door. open the red door. go to the green square."
24.07 ± 0.22	$+34.77$	"Pick up the red key after opening the grey door. Then walk to the red door, open it, and go to the goal."
17.57 ± 0.78	-1.63	"You cannot open the red door without a red key."
17.77 ± 0.54	-0.50	"Walking towards the red door is not very useful if it is closed."
—	—	"Go down until the second door on the left and pick up the key. Then exit the room and go down until the next door on the left and use it to open the door and get to the green box."
18.35 ± 1.93	$+2.76$	"Go to the room that has the red key, pick it up, and then go to the room with a red door. Enter the room, and go to the green goal object."

A Appendix

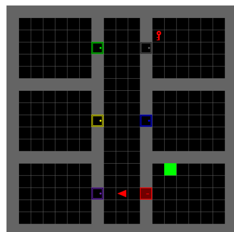

Figure 5: The initial state of the LockedRoom environment.

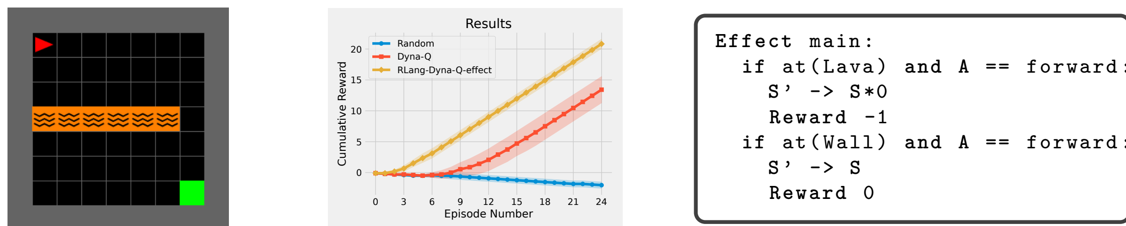

Figure 6: **LavaCrossing Experiment.** The agent was given the following advice: "Walking into lava will kill you. Walking into walls will do nothing." The initial state of LavaCrossing is pictured left, reward curves are in the center, and the grounded RLang advice is on the right.

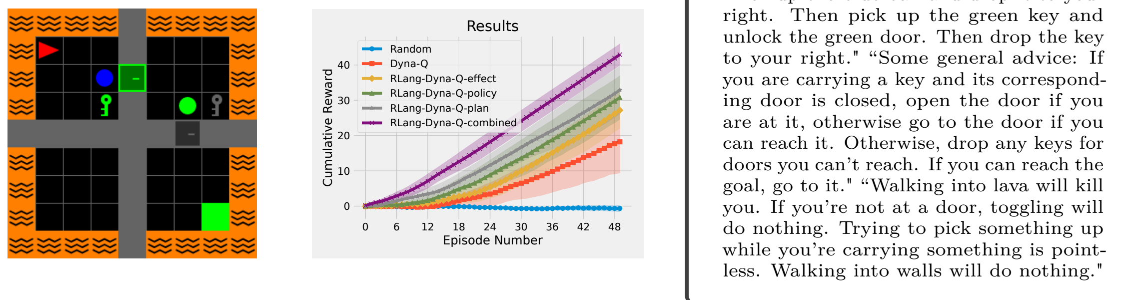

Figure 7: **MidMazeLava Experiment.** Language advice given to the agent was grounded to RLang effects, plans, and policies. The full translated RLang program is available in the appendix. All RLang-Dyna-Q agents outperformed Dyna-Q. The initial state of MidMazeLava is pictured on the left, reward curves are in the center, and the language advice given is on the right. The translated RLang program is in the appendix.

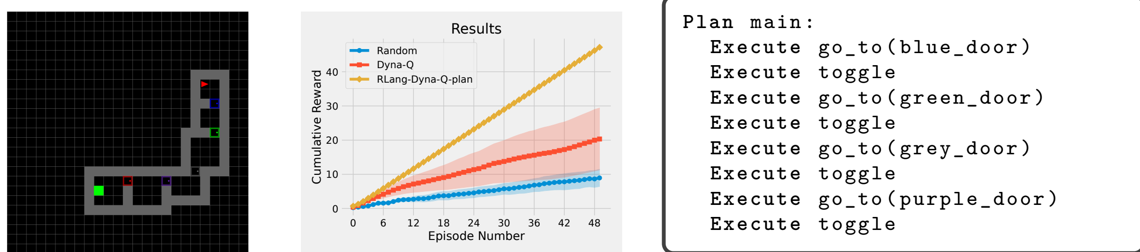

Figure 8: **MultiRoom Experiment.** The agent was given the following advice: "First go to the blue door, then the green door, then the grey door, then the purple door." The initial state of MultiRoom is pictured on the left, reward curves are center, and the translated advice is on the right.

A.1 Prompts Used for Translation Pipeline for Minigrid Experiments

A.1.1 [Minigrid] Prompt used for Stage 1 of the translation pipeline. Given a new piece of advice, we prompt the LLM to classify it as an Effect, Plan, or Policy.

RLang is a formal language for specifying information about every element of a Markov Decision Process (S,A,R,T). Each RLang object refers to one or more elements of an MDP. Here is a description of three important RLang groundings:

Policy: a direct function from states to actions, best used for more general commands.
Effect: a prediction about the state of the world or the reward function.
Plan: a sequence of specific steps to take.

Your task is to decide which RLang grounding most naturally corresponds to a given piece of advice:
Advice = "Don't touch any mice unless you have gloves on."
Grounding: Effect
Advice = "Walking into lava will kill you."
Grounding: Effect
Advice = "First get the money, then go to the green square."
Grounding: Plan
Advice = "Go through the door to the goal."
Grounding: Plan
Advice = "If you have the key, go to the door, otherwise you need to get the key."
Grounding: Policy
Advice = "If there are any closed doors, open them."
Grounding: Policy

A.1.2 [Minigrid] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang plan.

Your task is to translate natural language advice to RLang plan, which is a sequence of specific steps to take. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing plan function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use 'green_door' for the plan function.

Advice = "Open the door with the key and go through it to the goal"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'in_inventory']

```
Plan main:
  Execute go_to(yellow_key)
  Execute pickup
  Execute go_to(yellow_door)
  Execute toggle
  Execute go_to(goal)
```

Advice = "Get the key behind the red door to open the grey door. Then drop the key to the left."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'in_inventory']

```
Plan main:
  Execute go_to(red_door)
  Execute toggle
  Execute go_to(grey_key)
  Execute pickup
  Execute go_to(grey_door)
  Execute toggle
  Execute left
  Execute drop
```

A.1.3 [Minigrid] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang policy.

Your task is to translate natural language advice to RLang policy, which is a direct function from states to actions. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing policy function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use "green_door" for the policy function.

Advice = "If the yellow door is open, go through it and walk to the goal. Otherwise open the yellow door if you have the key."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'carrying']

```
Policy main:
  if yellow_door.is_open:
    Execute go_to(goal)
  elif carrying(yellow_key) and at(yellow_door) and not yellow_door.is_open:
    Execute toggle
```

Advice = "If you don't have the key, go get it."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'grey_key', 'red_door', 'grey_door', 'agent', 'purple_ball', 'at', 'carrying']

```
Policy main:
  if at(grey_key):
    Execute pickup
  elif not carrying(grey_key):
    Execute go_to(grey_key)
```

Advice = "If you are carrying a ball and its corresponding box is closed, open the box if you are at it, otherwise go to the box if you can reach it."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'green_ball', 'green_box', 'purple_box', 'agent', 'purple_ball', 'at', 'reachable', 'carrying']

```
Policy main:
  if carrying(green_ball) and not green_box.is_open:
    if at(green_box):
      Execute toggle
    elif reachable(green_box):
      Execute go_to(green_box)
```

Advice = "Drop any balls for boxes you can't reach"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'green_ball', 'green_box', 'purple_box', 'agent', 'purple_ball', 'at', 'reachable', 'carrying']

```
Policy main:
  if carrying(green_ball) and not reachable(green_box):
    Execute drop
  if carrying(purple_ball) and not reachable(purple_box):
    Execute drop
```

Advice = “if you have any key for a door that you cannot reach, you should drop it"
Primitives = [‘Agent’, ‘Wall’, ‘GoalTile’, ‘Lava’, ‘Key’, ‘Door’, ‘Box’, ‘Ball’, ‘left’, ‘right’, ‘forward’, ‘pickup’, ‘drop’, ‘toggle’, ‘done’, ‘pointing_right’, ‘pointing_down’, ‘pointing_left’, ‘pointing_up’, ‘go_to’, ‘step_towards’, ‘green_ball’, ‘green_box’, ‘purple_box’, ‘agent’, ‘purple_ball’, ‘at’, ‘reachable’, ‘carrying’]

```
Policy main:
  if carrying(green_key) and not reachable(green_door):
    Execute drop
  if carrying(purple_key) and not reachable(purple_door):
    Execute drop
  if carrying(red_key) and not reachable(red_door):
    Execute drop
```

Advice = “Hey listen, you can open the door if you have the key and at the door when the door is closed"
Primitives = [‘Agent’, ‘Wall’, ‘GoalTile’, ‘Lava’, ‘Key’, ‘Door’, ‘Box’, ‘Ball’, ‘left’, ‘right’, ‘forward’, ‘pickup’, ‘drop’, ‘toggle’, ‘done’, ‘pointing_right’, ‘pointing_down’, ‘pointing_left’, ‘pointing_up’, ‘go_to’, ‘step_towards’, ‘green_ball’, ‘green_box’, ‘purple_box’, ‘agent’, ‘purple_ball’, ‘at’, ‘reachable’, ‘carrying’]

```
Policy main:
  if carrying(purple_key) and not purple_door.is_open and at(purple_door):
    Execute toggle
```

A.1.4 [Minigrid] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang effect.

Your task is to translate natural language advice to RLang effect, which is a prediction about the state of the world or the reward function. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing effect function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use 'green_door' for the effect function.

Advice = "Don't go to the door without the key"
Primitives = ['yellow_door', 'goal', 'pickup', 'yellow_key', 'toggle', 'go_to', 'carrying', 'at']

```
Effect main:
  if at(yellow_door) and not carrying(yellow_key):
    Reward -1
```

Advice = "Don't walk into closed doors. If you're tired, don't go forward."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'green_ball', 'green_box', 'purple_box', 'agent', 'purple_ball', 'at', 'reachable', 'carrying']

```
Effect main:
  if at(yellow_door) and yellow_door.is_closed and A == forward:
    Reward -1
    S' -> S
  elif tired() and A == forward:
    Reward -1
```

Advice = "Walking into balls is pointless. You will die if you walk into keys. Trying to open a box when you aren't near it will do nothing."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'green_ball', 'green_box', 'purple_box', 'agent', 'purple_ball', 'at', 'reachable', 'carrying']

```
Effect main:
  if at(Ball) and A == forward:
    Reward 0
    S' -> S
  elif at(Key) and A == forward:
    Reward -1
    S' -> S*0
  elif at(Box) and A == toggle:
    Reward 0
    S' -> S
```

A.2 Prompts Used for Translation Pipeline for VirtualHome Experiments

A.2.1 [VirtualHome] Prompt used for Stage 1 of the translation pipeline. Given a new piece of advice, we prompt the LLM to classify it as an Effect, Plan, or Policy.

RLang is a formal language for specifying information about every element of a Markov Decision Process (S,A,R,T). Each RLang object refers to one or more elements of an MDP. Here is a description of three important RLang groundings:

Policy: a direct function from states to actions, best used for more general commands.
Effect: a prediction about the state of the world or the reward function.
Plan: a sequence of specific steps to take.

Your task is to decide which RLang grounding most naturally corresponds to a given piece of advice:
Advice = “Don’t touch any mice unless you have gloves on."
Grounding: Effect
Advice = “Walking into lava will kill you."
Grounding: Effect
Advice = “First get the money, then go to the green square."
Grounding: Plan
Advice = “Go through the door to the goal."
Grounding: Plan
Advice = “If you have the key, go to the door, otherwise you need to get the key."
Grounding: Policy
Advice = “If there are any closed doors, open them."
Grounding: Policy
Advice = “Open any doors if they are closed."
Grounding: Policy

A.2.2 [VirtualHome] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang plan.

Your task is to translate natural language advice to RLang plan, which is a sequence of specific steps to take. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing plan function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use 'green_door' for the plan function.

Advice = "Open the door with the key and go through it to the goal"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'in_inventory']

```
Plan main:
  Execute go_to(yellow_key)
  Execute pickup
  Execute go_to(yellow_door)
  Execute toggle
  Execute go_to(goal)
```

Advice = "Get the key behind the red door to open the grey door. Then drop the key to the left."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'in_inventory']

```
Plan main:
  Execute go_to(red_door)
  Execute toggle
  Execute go_to(grey_key)
  Execute pickup
  Execute go_to(grey_door)
  Execute toggle
  Execute left
  Execute drop
```

Advice = "Get the key behind the red door to open the grey door." Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'walk_to', 'open', 'close', 'putin', 'grab', 'inside', 'grey_key_11', 'red_door', 'grey_door_127', 'agent', 'purple_ball', 'is_on_a', 'at', 'at_any', 'in_inventory']

```
Plan main:
  Execute walk_to(red_door)
  Execute open(red_door)
  Execute walk_to(grey_key_11)
  Execute grab(grey_key_11)
  Execute walk_to(grey_door_127)
  Execute open(grey_door_127)
```

A.2.3 [VirtualHome] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang policy.

Your task is to translate natural language advice to RLang policy, which is a direct function from states to actions. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing policy function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use "green_door" for the policy function.

Advice = "If the yellow door is open, go through it and walk to the goal. Otherwise open the yellow door if you have the key."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'yellow_key', 'yellow_door', 'agent', 'goal', 'at', 'carrying']

```
Policy main:
  if yellow_door.is_open:
    Execute go_to(goal)
  elif carrying(yellow_key) and at(yellow_door) and not yellow_door.is_open:
    Execute toggle
```

Advice = "If you don't have the key, go get it"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'grey_key_11', 'red_door', 'grey_door', 'agent', 'purple_ball', 'is_on_a', 'at', 'at_any', 'in_inventory']

```
Policy main:
  if at(grey_key_11):
    Execute pickup
  elif not carrying(grey_key_11):
    Execute go_to(grey_key_11)
```

Advice = "If you're at the fridge, close it."
Primitives = ['Toothpaste', 'Bedroom', 'Character', 'Cereal', 'Bathroom', 'Sofa', 'Cabinet', 'Salmon', 'Pie', 'Kitchentable', 'Remotecontrol', 'Fridge', 'Microwave', 'Kitchen', 'Bookshelf', 'Livingroom', 'walk_to', 'open', 'close', 'putin', 'puton', 'grab', 'drop', 'can_drop', 'is_drop', 'inside', 'inside_something', 'on', 'at', 'is_closed', 'is_open', 'holding', 'near', 'character_1', 'kitchen_205', 'bookshelf_249', 'fridge_305', 'oven_133', 'pie_319', 'chicken_127', 'cabinet_19']

```
Policy main:
  if at(fridge_305):
    Execute close(fridge_305)
```

A.2.4 [VirtualHome] Prompt used for Stage 2 of the pipeline to translate a piece of advice into an RLang effect.

Your task is to translate natural language advice to RLang effect, which is a prediction about the state of the world or the reward function. For each instance, we provide a piece of advice in natural language, a list of allowed primitives, and you should complete the instance by filling the missing effect function. Don't use any primitive outside the provided primitive list corresponding to each instance, e.g., if there is no 'green_door' in the primitive list you must not use 'green_door' for the effect function.

Advice = "Don't go to the door without the key"
Primitives = ['yellow_door', 'goal', 'pickup', 'yellow_key', 'toggle', 'go_to', 'carrying', 'at']

```
Effect main:
  if at(yellow_door) and not carrying(yellow_key):
    Reward -1
```

Advice = "Don't walk into closed doors, since it takes no effect"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'pickup', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'go_to', 'step_towards', 'agent', 'goal', 'is_on_a', 'at', 'at_any', 'in_inventory']

```
Effect main:
  if at(yellow_door) and not yellow_door.is_open and A == forward:
    Reward -1
    S' -> S
```

Advice = "Walking to a broken object won't do anything. You can't grab the ball if it's inside something."
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'is_broken', 'left', 'right', 'forward', 'grab', 'drop', 'toggle', 'done', 'pointing_right', 'pointing_down', 'pointing_left', 'pointing_up', 'inside_something', 'go_to', 'step_towards', 'agent', 'goal', 'is_on_a', 'at', 'at_any', 'in_inventory', 'gate_12', 'door_16', 'ball_121']

```
Effect main:
  if A == walk_to(gate_12) and is_broken(gate_12):
    S' -> S
  if A == walk_to(door_16) and is_broken(door_16):
    S' -> S
  if A == grab(ball_121) and inside_something(ball_121):
    S' -> S
```

Advice = "Don't go to the purple ball"
Primitives = ['Agent', 'Wall', 'GoalTile', 'Lava', 'Key', 'Door', 'Box', 'Ball', 'left', 'right', 'forward', 'walk_to', 'open', 'close', 'putin', 'grab', 'inside', 'holding', 'grey_key_11', 'red_door', 'grey_door_127', 'agent', 'purple_ball', 'is_on_a', 'at', 'at_any', 'in_inventory']

```
Effect main:
  if A == walk_to(purple_ball):
    Reward -1
```

Advice = “If you put the pie into the microwave and the chicken into the oven, and make sure that they are both on, you will get reward and the episode will end."
Primitives = [’Toothpaste’, ’Bedroom’, ’Character’, ’Cereal’, ’Bathroom’, ’Sofa’, ’Cabinet’, ’Salmon’, ’Pie’, ’Kitchentable’, ’Remotecontrol’, ’Fridge’, ’Microwave’, ’Kitchen’, ’Bookshelf’, ’Livingroom’, ’walk_to’, ’open’, ’turn_on’, ’close’, ’putin’, ’puton’, ’grab’, ’drop’, ’can_drop’, ’is_drop’, ’inside’, ’inside_something’, ’on’, ’at’, ’is_closed’, ’is_open’, ’holding’, ’near’, ’character_1’, ’kitchen_205’, ’bookshelf_249’, ’fridge_305’, ’oven_133’, ’pie_319’, ’chicken_127’, ’microwave_19’]

```
Effect main:
  if inside(pie_319, microwave_19) and inside(chicken_127, oven_133):
    if is_closed(microwave_19) and at(oven_133) and A == turn_on(oven_133):
      Reward 5
      S’ -> S
    elif is_closed(oven_133) and at(microwave_19) and A == turn_on(microwave_19
  ):
      Reward 5
      S’ -> S
```

Advice = “If you’re not trying to pick up the fridge, you will be penalized" Primitives = [’Sofa’, ’Kitchentable’, ’Bathroom’, ’Salmon’, ’Kitchen’, ’Bookshelf’, ’Cereal’, ’Cabinet’, ’Livingroom’, ’Fridge’, ’Bedroom’, ’Character’, ’Toothpaste’, ’Pie’, ’Microwave’, ’Remotecontrol’, ’walk_to’, ’open’, ’close’, ’putin’, ’puton’, ’grab’, ’drop’, ’can_drop’, ’is_drop’, ’inside’, ’inside_something’, ’on’, ’at’, ’fridge_305’, ’is_pickup’, ’is_closed’, ’is_open’, ’holding’, ’near’, ’character_1’, ’bathroom_11’, ’toothpaste_62’, ’bedroom_73’, ’kitchen_205’, ’kitchentable_231’, ’bookshelf_249’, ’fridge_305’, ’microwave_313’, ’pie_319’, ’salmon_327’, ’cereal_334’, ’livingroom_335’, ’sofa_368’, ’cabinet_415’, ’remotecontrol_452’]

```
Effect main:
  if fridge_305(fridge_305) and not is_pickup(A):
    Reward -1
```

Advice = “if you have any key for a door that you cannot reach, you should drop it"
Primitives = [‘Agent’, ‘Wall’, ‘GoalTile’, ‘Lava’, ‘Key’, ‘Door’, ‘Box’, ‘Ball’, ‘left’, ‘right’, ‘forward’, ‘pickup’, ‘drop’, ‘toggle’, ‘done’, ‘pointing_right’, ‘pointing_down’, ‘pointing_left’, ‘pointing_up’, ‘go_to’, ‘step_towards’, ‘green_ball’, ‘green_box’, ‘purple_box’, ‘agent’, ‘purple_ball’, ‘at’, ‘reachable’, ‘carrying’]

```
Policy main:
  if carrying(green_key) and not reachable(green_door):
    Execute drop
  if carrying(purple_key) and not reachable(purple_door):
    Execute drop
  if carrying(red_key) and not reachable(red_door):
    Execute drop
```

Advice = “Hey listen, you can open the door if you have the key and at the door when the door is closed"
Primitives = [‘Agent’, ‘Wall’, ‘GoalTile’, ‘Lava’, ‘Key’, ‘Door’, ‘Box’, ‘Ball’, ‘left’, ‘right’, ‘forward’, ‘pickup’, ‘drop’, ‘toggle’, ‘done’, ‘pointing_right’, ‘pointing_down’, ‘pointing_left’, ‘pointing_up’, ‘go_to’, ‘step_towards’, ‘green_ball’, ‘green_box’, ‘purple_box’, ‘agent’, ‘purple_ball’, ‘at’, ‘reachable’, ‘carrying’]

```
Policy main:
  if carrying(purple_key) and not purple_door.is_open and at(purple_door):
    Execute toggle
```

A.3 User Study - Translated Advice for Minigrid Experiments

Table 3: Advice from the user study translated to RLang.

<table>
<tr><th>Language Advice</th><th>RLang Translation</th></tr>
<tr><td>“Remember to toggle to open doors."</td><td><pre>Policy main:
  if at(yellow_door) and not yellow_door.
  is_open:
    Execute toggle
  elif at(red_door) and not red_door.
  is_open:
    Execute toggle
  elif at(purple_door) and not purple_door.
  is_open:
    Execute toggle
  elif at(blue_door) and not blue_door.
  is_open:
    Execute toggle
  elif at(green_door) and not green_door.
  is_open:
    Execute toggle
  elif at(grey_door) and not grey_door.
  is_open:
    Execute toggle</pre></td></tr>
<tr><td>“You don’t need to carry keys to open the grey door."</td><td><pre>Effect main:
  if at(grey_door) and carrying(red_key):
    S’ -> S
    Reward -1
  elif at(grey_door) and carrying_something
  ():
    S’ -> S
    Reward -1</pre></td></tr>
<tr><td>“Identify the room with the red key, move to that room by opening the door. Pick up the key. Identify the room with the red door, proceed there. Open the red door. Find the green square and go there to finish the game."</td><td><pre>Plan main:
  Execute go_to(red_key)
  Execute pickup
  Execute go_to(red_door)
  Execute toggle
  Execute go_to(goal)</pre></td></tr>
<tr><td>“Move to the grey door, open it and enter the room until you get to the red key, pick it up. Exit the room and move towards the red door, open it and get into that room. Move to the green block and enter it."</td><td><pre>Plan main:
  Execute go_to(grey_door)
  Execute toggle
  Execute go_to(red_key)
  Execute pickup
  Execute go_to(grey_door)
  Execute toggle
  Execute go_to(red_door)
  Execute toggle
  Execute go_to(goal)</pre></td></tr>
<tr><td>“Go to the grey door. open the grey door. go to the red key. pick up the red key. go to the red door. open the red door. go to the green square."</td><td><pre>Plan main:
  Execute go_to(grey_door)
  Execute toggle
  Execute go_to(red_key)
  Execute pickup
  Execute go_to(red_door)
  Execute toggle
  Execute go_to(goal)</pre></td></tr>
</table>

Table 4: Advice from the user study translated to RLang (continued).

Language Advice	**RLang Translation**

"Pick up the red key after opening the grey door. Then walk to the red door, open it, and go to the goal."	`Plan main:`
	`  Execute go_to(grey_door)`
	`  Execute toggle`
	`  Execute go_to(red_key)`
	`  Execute pickup`
	`  Execute go_to(red_door)`
	`  Execute toggle`
	`  Execute go_to(goal)`

"You cannot open the red door without a red key."	`Effect main:`
	`  if at(red_door) and not carrying(red_key)`
	`  :`
	`    S' -> S`
	`    Reward -1`

"Walking towards the red door is not very useful if it is closed."	`Effect main:`
	`  if at(red_door) and not(red_door.is_open)`
	`   and A == forward:`
	`    S' -> S`
	`    Reward -1`

"Go down until the second door on the left and pick up the key. Then exit the room and go down until the next door on the left and use it to open the door and get to the green box."	`Plan main:`
	`  Execute go_to(second_left_door)`
	`  Execute pickup`
	`  Execute go_to(exit)`
	`  Execute go_to(next_left_door)`
	`  Execute toggle`
	`  Execute go_to(green_box)`

"Go to the room that has the red key, pick it up, and then go to the room with a red door. Enter the room, and go to the green goal object."	`Plan main:`
	`  Execute go_to(red_key)`
	`  Execute pickup`
	`  Execute go_to(red_door)`
	`  Execute toggle`
	`  Execute go_to(goal)`

A.4 MidMazeLava - Translated Advice

Advice: "Pick up the blue ball and drop it to your right. Then pick up the green key and unlock the green door. Then drop the key to your right. Some general advice: If you are carrying a key and its corresponding door is closed, open the door if you are at it, otherwise go to the door if you can reach it. Otherwise, drop any keys for doors you can't reach. If you can reach the goal, go to it. Walking into lava will kill you. If you're not at a door, toggling will do nothing. Trying to pick something up while you're carrying something is pointless. Walking into walls will do nothing."

```
Plan main:
  Execute go_to(blue_ball)
  Execute pickup
  Execute right
  Execute drop
  Execute go_to(green_key)
  Execute pickup
  Execute go_to(green_door)
  Execute toggle
  Execute right
  Execute drop

Policy main:
  if carrying(green_key) and not green_door.is_open:
    if at(green_door):
      Execute toggle
    elif reachable(green_door):
      Execute go_to(green_door)

  elif carrying(grey_key) and not grey_door.is_open:
    if at(grey_door):
      Execute toggle
    elif reachable(grey_door):
      Execute go_to(grey_door)

  elif reachable(goal):
    Execute go_to(goal)

  elif carrying(green_key) and not reachable(green_door):
    Execute drop

  elif carrying(grey_key) and not reachable(grey_door):
    Execute drop

Effect main:
  if at(Lava) and A == forward:
    S' -> S*0
    Reward -1
  if not at(Door) and A == toggle:
    S' -> S
    Reward 0
  if carrying_something() and A == pickup:
    S' -> S
    Reward 0
  if at(Wall) and A == forward:
    S' -> S
    Reward 0
```

A.5 HardMazeLight - Translated Advice

Advice: "Go and pick up the green ball, and drop it on your left, and then go pick up the blue key, and go to the blue door and open it up and drop the key on your left, and then go pick up the green key, and go to the green door to open it and drop the key on your left, and then go pick up the purple ball and drop it on your right. Nothing will happen if you walk towards the wall, or try to open a purple door without the purple key if it is locked. The applies for the yellow door and key as well as the red door and key. If you can reach the grey door and it is closed but you have the key, open it if you are at it or otherwise go to it. The same applies to the purple door, yellow door, and red door. Lastly, if you find the goal is reachable just go to the goal directly."

```
Plan main:
  Execute go_to(green_ball)
  Execute pickup
  Execute left
  Execute drop
  Execute go_to(blue_key)
  Execute pickup
  Execute go_to(blue_door)
  Execute toggle
  Execute left
  Execute drop
  Execute go_to(green_key)
  Execute pickup
  Execute go_to(green_door)
  Execute toggle
  Execute right
  Execute drop
  Execute go_to(purple_ball)
  Execute pickup
  Execute right
  Execute drop

Effect main:
  if at(Wall) and A == forward:
    Reward 0
    S' -> S
  elif at(purple_door) and purple_door.is_locked and A == toggle and not
  carrying(purple_key):
    Reward 0
    S' -> S
  elif at(yellow_door) and yellow_door.is_locked and A == toggle and not
  carrying(yellow_key):
    Reward 0
    S' -> S
  elif at(red_door) and red_door.is_locked and A == toggle and not carrying(
  red_key):
    Reward 0
    S' -> S
```

```
Policy main:
  if reachable(grey_door) and carrying(grey_key) and grey_door.is_locked:
    if at(grey_door):
      Execute toggle
    else:
      Execute go_to(grey_door)

  elif reachable(purple_door) and carrying(purple_key) and purple_door.
  is_locked:
    if at(purple_door):
      Execute toggle
    else:
      Execute go_to(purple_door)

  elif reachable(yellow_door) and carrying(yellow_key) and yellow_door.
  is_locked:
    if at(yellow_door):
      Execute toggle
    else:
      Execute go_to(yellow_door)

  elif reachable(red_door) and carrying(red_key) and red_door.is_locked:
    if at(red_door):
      Execute toggle
    else:
      Execute go_to(red_door)

  elif reachable(goal):
    Execute go_to(goal)
```

A.6 FoodSafety - Translated Advice

Advice: “Go to fridge and open it, and then go find the pie and pick it up, walk back to the fridge and put the pie in the fridge. You have to close the fridge too", “If the salmon is in the microwave, and you are at the microwave and it’s open, close it. Otherwise if you are holding salmon, do the following: open the microwave if you are near it but it’s closed, put the salmon into the microwave if it’s open and you’re near it, else walk to the microwave.", “If the pie is in the fridge, and the salmon is in the microwave, then closing the fridge if the microwave is closed or closing the microwave if the fridge is closed will give you reward and end the episode."

```
Plan main:
  Execute walk_to(fridge_305)
  Execute open(fridge_305)
  Execute walk_to(pie_319)
  Execute grab(pie_319)
  Execute walk_to(fridge_305)
  Execute putin(fridge_305)
  Execute close(fridge_305)


Policy main:
  if inside(salmon_327, microwave_313) and at(microwave_313) and is_open(
  microwave_313):
    Execute close(microwave_313)
  elif holding(salmon_327):
    if at(microwave_313) and is_closed(microwave_313):
      Execute open(microwave_313)
    elif at(microwave_313) and is_open(microwave_313):
      Execute putin(microwave_313)
    else:
      Execute walk_to(microwave_313)

Effect main:
  if inside(pie_319, fridge_305) and inside(salmon_327, microwave_313):
    if is_closed(fridge_305) and at(microwave_313) and A == close(microwave_313
  ):
      Reward 5
      S' -> S
    elif is_closed(microwave_313) and at(fridge_305) and A == close(fridge_305)
  :
      Reward 5
      S' -> S
```

A.7 CouchPotato - Translated Advice

Advice: “If you’re holding the toothpaste and can drop it, drop it.", “Go grab the remote control and put it on the sofa.", “If you’re holding the toothpaste and not trying to drop it, you will be penalized. Also, nothing will happen if you try to walk to the remote control, cereal, toothpaste, or salmon, if you try to walk to them and they are contained inside anything."

```
Effect main:
  if holding(toothpaste_62) and not is_drop(A):
    Reward -1
  if inside_something(remotecontrol_452) and A == walk_to(remotecontrol_452):
    S' -> S
  if inside_something(cereal_334) and A == walk_to(cereal_334):
    S' -> S
  if inside_something(toothpaste_62) and A == walk_to(toothpaste_62):
    S' -> S
  if inside_something(salmon_327) and A == walk_to(salmon_327):
    S' -> S


Policy main:
  if holding(toothpaste_62) and can_drop(toothpaste_62):
    Execute drop(toothpaste_62)

Plan main:
  Execute walk_to(remotecontrol_452)
  Execute grab(remotecontrol_452)
  Execute walk_to(sofa_368)
  Execute puton(remotecontrol_452, sofa_368)
```

B Additional Experiment: Grounding Commands to RLang Plans

We compare our method to SayCan (Ahn et al., 2022), which uses the commonsense reasoning capacity of LLMs to satisfy a natural language request by generating a simple plan consisting of a series of pre-engineered high-level robot skills. Adopting the same

In this experiment we demonstrate that, in a simulated 3-dimensional physical environment, RLang can express the full range of natural language instructions necessary for a robot to complete various tasks. By grounding natural language instructions to RLang policies over this environment, we achieve performance on par with the results from the open-source tasks that the original SayCan paper evaluated on, showing that RLang can be easily substituted for the formal language that the SayCan authors developed for this specific task, allowing for generalization without sacrificing performance.

Similar to the SayCan work, we assume that we are given a grounding tuple $\langle \Pi, S, A, \rangle$, and a set of skills Π, where each skill $\pi \in \Pi$ performs an action with the robot arm to manipulate a block or a bowl. We evaluate on the 8 unique tasks made available in the open-source version of SayCan, running each task across 10 different randomly selected initial states, using both the native SayCan language and RLang as the DSL for grounding natural language instructions to robot behavior.

Each task configuration that the original SayCan agent completes, is also completed by the RLang agent. While their behavior on failure cases occasionally varied, these were generally caused by errors in the vision model’s processing of shadows in the simulated environment. These generally caused the textual scene description fed into GPT-3 to include a block where a bowl should be, and occasionally incorrect color labels, which often provided the text-only planner with a nonsensical task that was impossible to complete. Similarly, in cases where multiple action orders could satisfy the request, the RLang and SayCan pipelines occasionally diverged in the order of actions. Nonetheless, neither language grounding pipeline completed a task configuration that the other one did not.

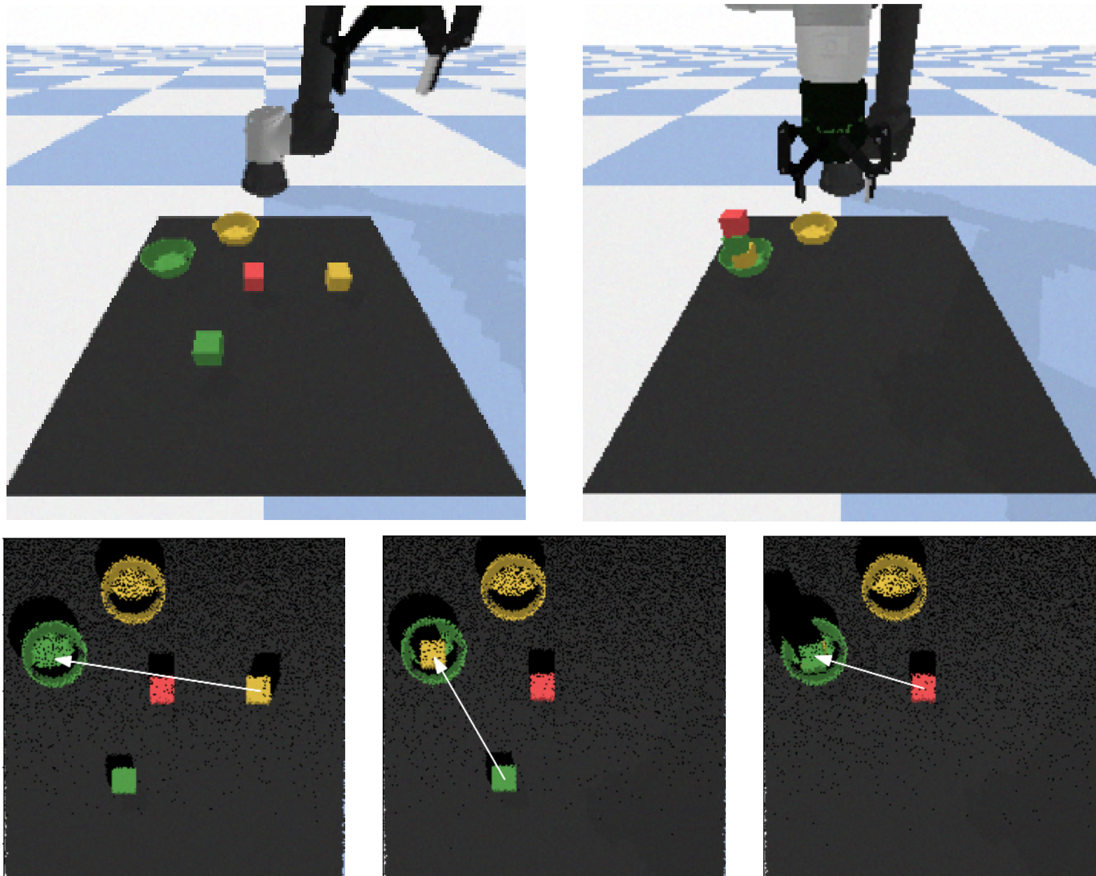

Figure 9: Top: One configuration of the initial and completed states in the SayCan environment. Bottom: the action sequence to execute on the instruction: "put all the blocks in the green bowl.", from the robot arm's perspective

Table 5: Success rates of SayCan and RLang-based instruction grounding rate on each task, out of 10 random initial states.

Instruction	SayCan	NL2RLang
put all the blocks in different corners.	10	10
move the block to the bowl.	6	6
put any blocks on their matched colored bowls.	7	7
put all the blocks in the green bowl.	7	7
stack all the blocks.	8	8
make the highest block stack.	7	7
put the block in all the corners.	10	10
clockwise, move the block through all the corners.	10	10

C Visual grounding experiments in VirtualHome

D Experiment Parameters

In all experiments, for both Dyna-Q and RLang-Dyna-Q, we set the learning rate α to 0.1, the discount factor γ to 0.99, $\epsilon_1 = \epsilon_2 = 0.1$, except when there is policy or plan advice, uniform exploration, and 16 hallucinatory updates with the learned dynamics model.

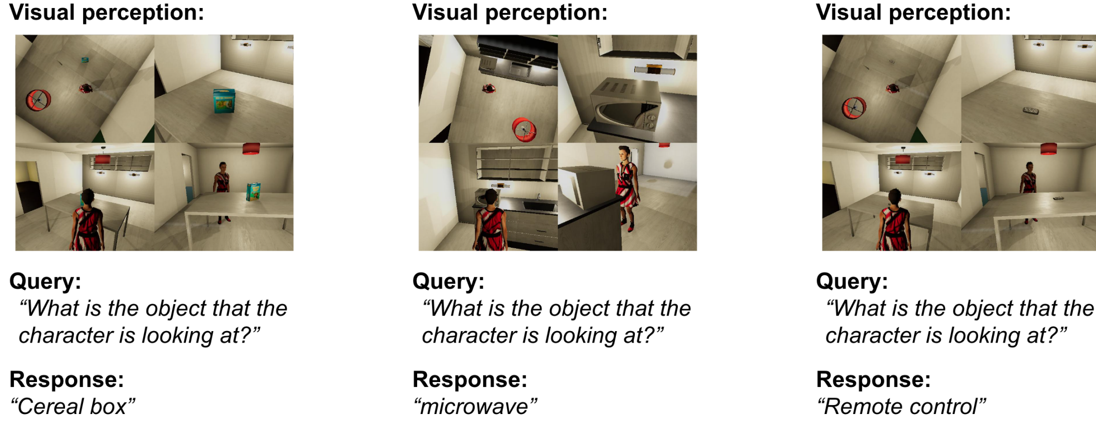

Figure 10: By programmatically interacting with unnamed objects in the environment, we can capture images from different perspectives of each object, which are collectively seamed into one combined figure and fed to the VLM with an explicit natural language query regarding the label of the object. Then, the labels output by VLM will be provided to the translation pipeline as primitives for grounding natural language advice into RLang.

Algorithm 1 RLang-Dyna-Q Agent

Given: π_{RLang}, T_{RLang}, R_{RLang} from an RLang program
Init $Q(s,a)$, $T(s,a)$, $R(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
loop
  $s \leftarrow$ current (nonterminal) state
  $a \leftarrow \epsilon_1, \epsilon_2$-greedy$(s, \pi_{\text{RLang}}, Q)$ # With prob. ϵ_2, we execute the RLang plan or policy
  Execute action a; observe next state s', and reward r
  $Q \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  $T(s,a), R(s,a) \leftarrow s', r$ # Update our model
  for $i = 1$ to N_1 **do**
    $s \leftarrow$ random previously observed state
    $a \leftarrow$ random action previously taken in s
    $s', r \leftarrow T(s,a), R(s,a)$
    $Q \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  end for
  for $i = 1$ to N_2 **do**
    $s \leftarrow$ random previously observed state
    $a \leftarrow$ random action **not** previously taken in s
    $s', r \leftarrow T_{\text{RLang}}(s,a), R_{\text{RLang}}(s,a)$ Predict s', r using dynamics given by RLang
    $Q \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  end for
end loop

For the translation step, we use the `gpt-3.5-turbo-instruct` model with a temperature of 0.