PROGRAM GUIDED AGENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Developing agents that can learn to follow natural language instructions has been an emerging research direction. While being accessible and flexible, natural language instructions can sometimes be ambiguous even to humans. To address this, we propose to utilize programs, structured in a formal language, as a precise and expressive way to specify tasks. We then devise a modular framework that learns to perform a task specified by a program – as different circumstances give rise to diverse ways to accomplish the task, our framework can perceive which circumstance it is currently under, and instruct a multitask policy accordingly to fulfill each subtask of the overall task. Experimental results on a 2D Minecraft environment not only demonstrate that the proposed framework learns to reliably accomplish program instructions and achieves zero-shot generalization to more complex instructions but also verify the efficiency of the proposed modulation mechanism for learning the multitask policy. We also conduct an analysis comparing various models which learn from programs and natural language instructions in an end-to-end fashion.

1 INTRODUCTION

Humans are capable of leveraging instructions to accomplish complex tasks. A comprehensive instruction usually comprises a set of descriptions detailing a variety of situations and the corresponding subtasks that are required to be fulfilled. To accomplish a task, we can leverage instructions to estimate the progress, recognize the current state, and perform corresponding actions. For example, to make a gourmet dish, we can follow recipes and procedurally create the desired dish by recognizing what ingredients and tools are missing, what alternatives are available, and what corresponding preparations are required.

Can machines likewise learn to follow and exploit comprehensive instructions like humans? Utilizing expert demonstrations to instruct agents has been widely studied in (Finn et al., 2017; Yu et al., 2018; Xu et al., 2018; Pathak et al., 2018; Stadie et al., 2017; Duan et al., 2017; Wang et al., 2017). However, demonstrations could be expensive to obtain and are less flexible (*e.g.* altering subtask orders in demonstrations is nontrivial). On the other hand, natural language instructions are flexible and expressive (Misra et al., 2018; Fried et al., 2018; Kaplan et al., 2017; Bahdanau et al., 2019). Yet, language has the caveat of being ambiguous even to humans, due to its lacking of structure as well as unclear coreferences and entities. Andreas et al. (2017a); Oh et al. (2017) investigate hierarchical approaches, where an instruction consists of a set of symbolically represented subtasks. Nonetheless, those instructions are not a function of states (*i.e.* describe a variety of circumstances and the corresponding desired subtasks), which substantially limits their expressiveness.

We propose to utilize *programs*, written in a formal language, as a structured, expressive, and unambiguous representation to specify tasks. Specifically, we consider programs, which are composed of control flows (*e.g.* if/else and loops), environmental conditions, as well as corresponding subtasks, as shown in Figure 1. Not only do programs have expressiveness by describing diverse situations (*e.g.* a river exists) and the corresponding subtasks which are required to be executed (*e.g.* mining wood), but they are also unambiguous due to their explicit scoping. To study the effectiveness of using programs as task specifications, we introduce a new problem, where we aim to develop a framework which learns to comprehend a task specified by a program as well as perceive and interact with the environment to accomplish the task.

To address this problem, we propose a modular framework, *program guided agent*, which exploits the structural nature of programs to decompose and execute them as well as learn to ground pro-

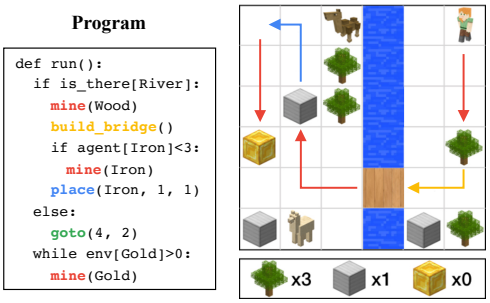


Figure 1: An illustration of the proposed problem. We are interested in learning to fulfill tasks specified by written programs. A program consists of control flows (*e.g.* `if`, `while`), branching conditions (*e.g.* `is_there[River]`), and subtasks (*e.g.* `mine(Wood)`).

Program p := def run() : s
Statement s := while(c) : (s) | b | loop(r) : (s)
| if(c) : (s) | elseif(c) : (s) | else : (s)
Repetition r := Number of repetitions
Item t := Gold | Wood | Iron
Operators o := $> \geq == < \leq$
Numbers i := Positive Integers
Perception h := agent[] | env[] | is_there[]
Behavior b := mine(t) | goto(i, i)
| place(t, i, i) | build_bridge() | sell(t)
Conditions c := $h[t]$ o i

Figure 2: The domain-specific language (DSL) for constructing the programs. Each program is composed of domain dependent perception, subtasks, and control flows.

gram tokens with the environment. Specifically, our framework consists of three modules: (1) a program interpreter that leverages grammar provided by the programming language to parse and execute a program, (2) a perception module that learns to respond to conditional queries (*e.g.* `is_there[River]`) produced by the interpreter, and (3) a policy that learns to fulfill a variety of subtasks (*e.g.* `mine(Wood)`) extracted from the program by the interpreter. To effectively instruct the policy with symbolically represented subtasks, we introduce a learned modulation mechanism that leverages a subtask to modulate the encoded state features instead of concatenating them. Our framework (shown in Figure 3) utilizes a *rule-based* program interpreter (*i.e.* a compiler) to deal with programs as well as *learning* perception module and policy when it is necessary to perceive or interact with the environment. With this modularity, our framework can generalize to more complex program-specified tasks without additional learning.

To evaluate the proposed framework, we consider a Minecraft-inspired 2D gridworld environment, where an agent can navigate itself across different terrains and interact with objects, similar to Andreas et al. (2017a); Oh et al. (2017); Sohn et al. (2018). A corresponding domain-specific language (DSL) defines the rules of constructing programs for instructing an agent to accomplish certain tasks. Our proposed framework demonstrates superior generalization ability – learning from simpler tasks while generalizing to complex tasks. We also conduct extensive analysis on various end-to-end learning models which learns from not only program instructions but also natural language descriptions. Furthermore, our proposed learned policy modulation mechanism yields a better learning efficiency compared to other commonly used methods that simply concatenate a state and goal.

2 Related Work

Learning from language instructions. Prior works have investigated leveraging natural languages to specify tasks on a wide range of applications, including navigation (Misra et al., 2018; Tellex et al., 2011; Fried et al., 2018), spatial reasoning for goal reaching (Janner et al., 2018), game playing (Kaplan et al., 2017; Fried et al., 2017; Co-Reyes et al., 2019), and grounding visual concepts (Kaplan et al., 2017; Bahdanau et al., 2019; Andreas et al., 2017b). However, natural language descriptions can often be ambiguous even to humans. Moreover, it is not clear how end-to-end learning agents trained with simpler instructions can generalize well to much more complex ones. In contrast, we propose to utilize an precise and structured representation, programs, to specify tasks.

Learning from demonstrations. When a task cannot be easily described in languages (*e.g.* object texture or geometry), expert demonstrations offer an alternative way to provide instructions. Prior works have explored learning from video demonstrations (Finn et al., 2017; Yu et al., 2018; Xu et al., 2018; Pathak et al., 2018; Stadie et al., 2017) or expert trajectories (Duan et al., 2017; Wang et al., 2017). However, demonstrations can be expensive to obtain and are less expressive about diverging behaviors of a complex task, which are better captured by control flow in programs. Moreover, editing demonstrations such as rearranging the order of subtasks is often difficult.

Program induction and synthesis. To learn acquire programmatic skills such as digit addition and string transformations and achieve better generalization, program induction methods (Xu et al., 2018; Devlin et al., 2017a; Neelakantan et al., 2015; Graves et al., 2014; Kaiser & Sutskever, 2016; Reed & De Freitas, 2016; Cai et al., 2017) aim to implicitly induce the underlying programs to mimic the behaviors demonstrated in task specifications (*e.g.* input/output pairs, demonstrations). On the other hand, program synthesis methods (Parisotto et al., 2017; Devlin et al., 2017b; Chen et al., 2019; Shin et al., 2018; Bunel et al., 2018; Liu et al., 2019; Sun et al., 2018; Lin et al., 2018; Liao et al., 2019) explicitly synthesize the underlying programs and execute the programs to perform the tasks. Instead of trying to infer programs from task specifications, we are interested in explicitly reading and executing programs. Once learned, our framework can be leveraged to obtain program execution results for evaluating program synthesis frameworks when no program executor is available (*e.g.* programs describe real-world activities instead of behaviors in simulation).

Programmable agent. Executing programs with reinforcement learning have been studied in programmable hierarchies of abstract machines (PHAM) (Parr & Russell, 1998; Andre & Russell, 2001; 2002), which provide partial descriptions and subroutines of the desired task. Programs can also take the form of a graph to express subtask dependencies and different routes of completing a task (Sohn et al., 2018). Agents can also be trained to execute declarative programs by grounding these well-structured languages in their learning environments (Denil et al., 2017; Lázaro-Gredilla et al., 2018). In contrast, our modular framework consists of modules for perceiving the environment and interacting with it by following a program which specifies the task.

3 Problem Formulation

We are interested in learning to comprehend and execute an instruction specified by a program to fulfill the desired task. In this section, we formally describe our definition of programs, the family of Markov Decision Processes (MDPs), and the problem formulation.

Program. The programs considered in this work are defined based on a domain-specific language (DSL) as shown in Figure 2. The DSL is composed of perception primitives, action primitives, and control flow. A perception primitive indicates circumstances in the environment (*e.g.* `is_there(River)`, and `agent[Gold]<3`) that can be perceived by an agent, while an action primitive defines a subtask that describes a certain behavior (*e.g.* `mine(Gold)`, and `goto(1,1)`). Control flow includes `if/else` statements, loops, and Boolean/logical operators to compose more sophisticated conditions. A program p is a deterministic function that outputs a desired behavior (*i.e.* subtask) given a history of states $o_t = p\left(H_j\right)$, where $H_j = \{s_1, ..., s_t\}$ is a state history with $s \in \mathcal{S}$ denoting a state of the environment, and $o \in \mathcal{O}$ denotes an instructed behavior (subtask). We denote a program as $p \sim \mathcal{P}$, an infinite program set containing all executable programs given a DSL.

MDPs. We consider a family of finite-horizon discounted MDPs in a shared environment, specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{T}, \mathcal{R}, \rho, \gamma)$, where $\mathcal{S}$ denotes a set of states, $\mathcal{A}$ denotes a set of low-level actions an agent can take, $\mathcal{P}$ denotes a set of programs specifying instructions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ denotes a transition probability distribution, $\mathcal{R}$ denotes a task-specific reward function, ρ denotes an initial state distribution, and γ denotes a discount factor. For a fixed sequence $\{(s_0, a_0), ..., (s_t, a_t)\}$ of states and actions obtained from a rollout of a given policy π, the performance of the policy is evaluated based on a discounted return $\sum_{t=0}^{T} \gamma^t r_t$, where T is the horizon of the episode.

Problem Formulation. We consider developing a framework which can comprehend and fulfill an instruction specified by a program. Specifically, we consider a sampled MDP with a program describing the desired task. Addressing this task requires the ability to keep track of which parts of the program are finished and which parts are not, perceiving the environment and deciding which paths in the program to take, and performing actions interacting with the environment to fulfill subtasks.

4 Approach

Accomplishing an instructed task described by a program requires (1) comprehending the program control flow and conditions, (2) recognizing the situations to infer which path in the program should be chosen, and (3) executing a series of actions interacting with the environment to fulfill the subtasks. Based on this intuition, we design a modular framework with three modules:

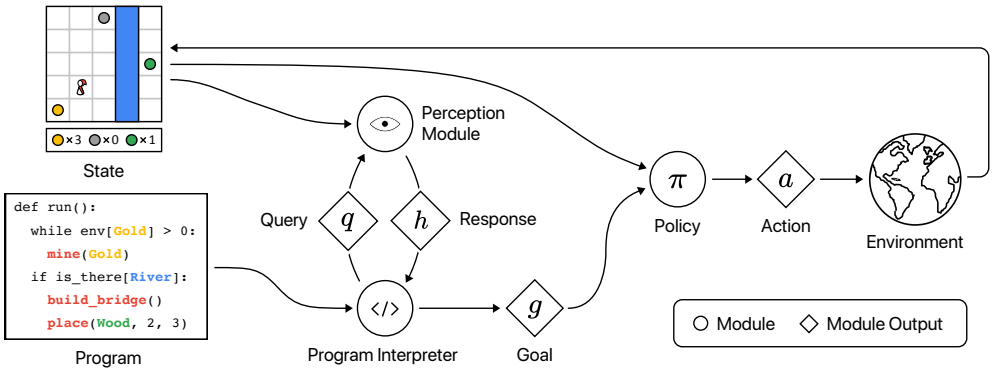


Figure 3: Program Guided Agent. The proposed modular framework comprehends and fulfills a desired task specified by a program. The program interpreter executes the program by altering between querying the perception module with a query q when an environment condition is encountered (*e.g.* `env[Gold]>0`, `is_there[River]`) and instructing a policy when it needs to fulfill a goal/subtask g (*e.g.* `mine(Gold)`, `build_bridge()`). The perception module produces a response h to answer the query, determining which paths in the program should be chosen. The policy takes a sequence of low-level actions a (*e.g.* `moveUp`, `moveLeft`, `Pickup`) interacting with the environment to accomplish the given subtask (*e.g.* `mine(Gold)`).

- **Program interpreter** (Section 4.1) reads a program and executes it by querying a perception module with environment conditions (*e.g.* `env[Gold]>0`) and instructing the policy with subtasks (*e.g.* `mine(Gold)`).
- **Perception module** (Section 4.2) responds to perception queries (*e.g.* `env[Gold]>0`) by examining the observation and predicting responses (*e.g.* `true`).
- **Policy (action module)** (Section 4.3) performs low-level actions (*e.g.* `moveUp`, `moveLeft`, `pickUp`) to fulfill the symbolically represented subtasks (*e.g.* `mine(Gold)`) provided by the program interpreter.

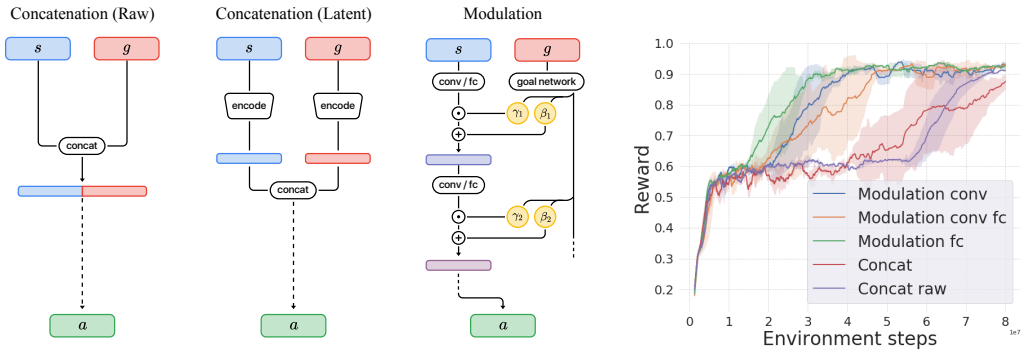
Our key insight is to only *learn* a module when its input or output is associated with the environment (*i.e.* a function approximator is needed) – the perception module learns to ground the queries to its observation and answer them; the policy learns to ground the symbolically represented subtasks and interact with the environment in a trial-and-error way (Section 4.4). On the other hand, we do not learn the program interpreter; instead, we utilize a *rule-based* parser to execute programs (*i.e.* compiler). An overview of the proposed framework is illustrated in Figure 3.

4.1 Program Interpreter

To comprehend a program instruction, we group program tokens into three main categories: (1) **subtasks** indicate what the agent should perform (*e.g.* `mine(Gold)`), (2) **perceptions** the essential information extracted from the environment (*e.g.* `env[Gold]>0`), and (3) **control flows** determine which paths in a program should be taken according to the perceived information (*i.e.* perceptions). Then, we devise a *program interpreter*, which can execute and keep track of the progress by leveraging the structure of programs. Specifically, it consists of a program line parser and a program executor. The parser first transforms the program into a program tree by representing each line of a program as a tree node. Each node is either a leaf node (subtask) or a non-leaf node (perception or control flow) that has various subroutines as children nodes. The executor then performs a pre-order traversal on the program tree to execute the program, utilizing the parsed contents to alternate between querying the perception module when an environment condition is encountered and instructing the policy when it reaches to a leaf node (subtask). The details and the algorithm are summarized in section A. Note that the *program interpreter* is a rule-based algorithm instead of a learning module.

4.2 Perception Module

Determining which paths should be chosen when executing a program requires grounding a symbolically represented query (*e.g.* `is_there[River]` can be represented as a sequence of symbols) and perceiving the environment. To this end, we employ a *perception module* Φ that learns to map a



(a) Illustration (b) Training curves

Figure 4: Learning a multitask policy via learned modulation. (a) A multitask policy takes both a state s and a goal specification g as inputs and produces an action distribution $a \sim \pi(s, g)$. Instead of simply concatenating the state and goal in a raw space or a latent space, we propose to modulate state features e_s using the goal. Specifically, the goal network learns to predict affine transform parameters γ and β to modulate the state features $\hat{e}_s = \gamma \cdot e_s + \beta$. Then, the final layers use the modulated features to predict actions. (b) We experiment different ways of feeding a state and goal for learning a multitask policy. The training curves demonstrate that all modulation variants, including modulating state feature maps of convolutional layers (*Modulation conv*), modulating state feature vectors of fully-connected layers (*Modulation fc*), or both (*Modulation conv fc*), are more efficient than concatenating the state and the goal in a raw space (*Concat raw*) or a latent space (*Concat*).

query and current observation to a response: $h = \Phi(q, s)$, where q denotes a query, and h denotes the corresponding perception output (*e.g.* `true/false`). Note that we focus on Boolean perception outputs in this paper, but a more generic perception type can be used (*e.g.* object attributes such as color, shape, and size).

4.3 Policy

When program execution reaches a subtask/leaf node (*e.g.* `mine(Gold)`), the agent is required to take a sequence of low-level actions (*e.g.* `moveUp`, `moveLeft`, `Pickup`) to interact with the environment to fulfill it. To enable the execution, we employ a multitask policy π (*i.e.* action module) which is instructed by a symbolic goal (*e.g.* `mine(Gold)`) provided by the program interpreter indicating the details of the corresponding subtask. To learn to perform different subtasks, we train the policy using actor-critic reinforcement learning, which takes a goal vector g and an environment state s and outputs a probabilistic distribution a for low-level actions $a \sim \pi(s_t, g_t|\theta)$. The value estimator used for our policy optimization is also goal-conditioned: $V_\pi(s_t, g_t) = \mathbb{E}[\sum_t \gamma^t R_t | s_0 = s, \pi, g_t]$.

While the most common way to feed a state and goal to a policy parameterized by a neural network is to concatenate them in a raw space or a latent space, we find this less effective when the policy has to learn a diverse set of tasks. Therefore, we propose a modulation mechanism to effectively learn the policy. Specifically, we employ a goal network to encode the goal and compute affine transform parameters γ and β, which are used to modulate state features e_s to $\hat{e}_s = \gamma \cdot e_s + \beta$. Then, the modulated features $\hat{e}_s$ are used to predict action a and value V. With the modulation mechanism, the goal network learns to activate state features related to the current goal and deactivate others. An illustration is shown in Figure 4 (a).

4.4 Learning

To follow a program by perceiving the environment and taking actions to interact with it, we employ two learning modules: a perception module and a policy. In this section, we discuss how each module is trained, their training objectives, and optimization methods. More training details and the architectures can be found in section B.3.1.

4.4.1 PERCEPTION MODULE

We formulate training the perception module as a supervised learning task. Given tuples of (query q, state s, ground truth perception h_{gt}), we train a neural network Φ to predict the perception output h by optimizing the binary cross-entropy loss: $\mathcal{L}_{CE} = -h_{gt}log(h) - (1 - h_{gt})log(1 - h)$. Note that when perception describes more than a Boolean, training the perception module can be done by optimizing other losses such as categorical cross-entropy loss. We train the perception module only on the queries appearing in the training programs with randomly sampled states, requiring it to generalize to novel queries to perform well in executing testing programs.

4.4.2 POLICY

We train the policy using Advantage Actor-Critic (A2C) (Mnih et al., 2016; Dhariwal et al., 2017), which is commonly used for gridworld environments with discrete action spaces. A2C computes policy gradients $A_t \nabla_\theta \log \pi_\theta (a_t|s_t, g_t)$, where $A_t = R_t - V(s_t, g_t)$ is the advantage function based on empirical return R_t starting from s_t and learned value estimator $V(s_t, g_t)$ conditioning on the goal vector g_t. We denote the learning rate as α, and the policy update rule is as follows:

$$\theta \leftarrow \theta + \alpha \left(A_t \nabla_\theta \log \pi_\theta (a_t|s_t, g_t) + \beta \nabla_\theta H_{\pi_\theta} \right), \tag{1}$$

where H_{π_θ} denotes the policy entropy, where maximizing it improves overall exploration, and β determines the strength of the entropy regularization term.

5 EXPERIMENTS

Our experiments aim to answer the following questions: (1) Can our proposed framework learn to perform tasks specified by programs? (2) Can our modular framework generalize better to more complex tasks compared to end-to-end learning models? (3) How well can a variety of end-to-end learning models (*e.g.* LSTM, Tree-RNN, Transformer) learn from programs and natural language instructions? (4) Is the proposed learned modulation more efficient to learn a multitask (multi-goal) policy than simply concatenating a state and goal?

5.1 EXPERIMENTAL SETUPS

5.1.1 ENVIRONMENT

To evaluate the proposed framework in an environment where an agent can perceive diverse scenarios and interact with the environment to perform various subtasks, we construct a discrete Minecraft-inspired gridworld environment, similar to Andreas et al. (2017a); Oh et al. (2017); Sohn et al. (2018). As illustrated in Figure 1, the agent can navigate through a grid world and interact with resources (*e.g.* Wood, Iron, Gold) and obstacles (*e.g.* River, Wall), build tools (*e.g.* Bridge), and sell resources to a merchant visualized as an alpaca. The environment gives a sparse task completion reward of $+1$ when an instruction (*i.e.* an entire program or natural language instruction) is successfully executed. More details can be found in Section B.1.

5.1.2 TASK INSTRUCTIONS

Programs. We sample 4,500 programs using our DSL and split them into 4,000 training programs (*train*) and 500 testing programs (*test*). To examine the framework's ability to generalize to more complex instructions, we generate 500 programs which are twice longer and contains more condition branches on average to construct a harder testing set (*test-complex*).

Natural language instructions. To obtain the natural language counterparts of those instructions, we asked annotators to construct natural language translations of all the programs. The data collection details, as well as sample programs and their corresponding natural language translations, can be found in Section B.2.

5.2 TRAINING

During training, we randomly sample programs from the training set as well as randomly sample an environment state to execute the program interpreter. The program interpreter produces a goal to

Table 1: Task completion rate. For each method, we iterate over all the programs in a testing set by randomly sampling ten initial environment states and running three models trained using different random seeds for this method. The averaged task completion rates and their standard deviations are reported. Note that all the end-to-end learning models learning from natural language descriptions and programs suffer from a significant performance drop when evaluated on the more complex testing set.

Instruction		**Natural language descriptions**		**Programs**				
Method		Seq-LSTM	Transformer	Seq-LSTM	Tree-RNN	Transformer	Ours (concat)	Ours
Dataset	test	54.9±1.8%	52.5±2.6%	56.7±1.9%	50.1±1.2%	49.4±1.6%	88.6±0.8%	94.0±0.5%
	test-complex	32.4±4.9%	38.2±2.6%	38.8±1.2%	42.2±2.4%	40.9±1.5%	85.2±0.8%	91.8±0.2%
Generalization gap		40.9%	27.2%	31.6%	15.8%	17.2%	3.8%	2.3%

instruct the policy when encountering a subtask in the program. The policy takes actions $a \sim \pi(s, g)$ and receive reward $+1$ only when the entire program is completed. This setup naturally induces a curriculum where the policy first learns to solve simpler programs and obtain a better understanding of the subtasks, which eventually allows the policy to complete more complex programs. Note that the perception module is pre-trained beforehand in a supervised manner. More training details can be found in section B.5.

5.3 End-to-end learning models

In contrast to the proposed modular framework, we experiment with a variety of end-to-end learning models. Considering programs and natural language instructions as sequences of tokens, we investigate two types of state-of-the-art sequence encoders: LSTM (Hochreiter & Schmidhuber, 1997) (Seq-LSTM), and Transformers (Vaswani et al., 2017; Devlin et al., 2018) (Transformer). To leverage the explicit structure of programs, we also investigate encoding programs using a generalization of RNNs for tree-structured input (Tai et al., 2015; Alon et al., 2019) (Tree-RNN). All the models are trained using A2C. The details of these architectures can be found in Section B.3.2.

5.4 Results

5.4.1 Task completion

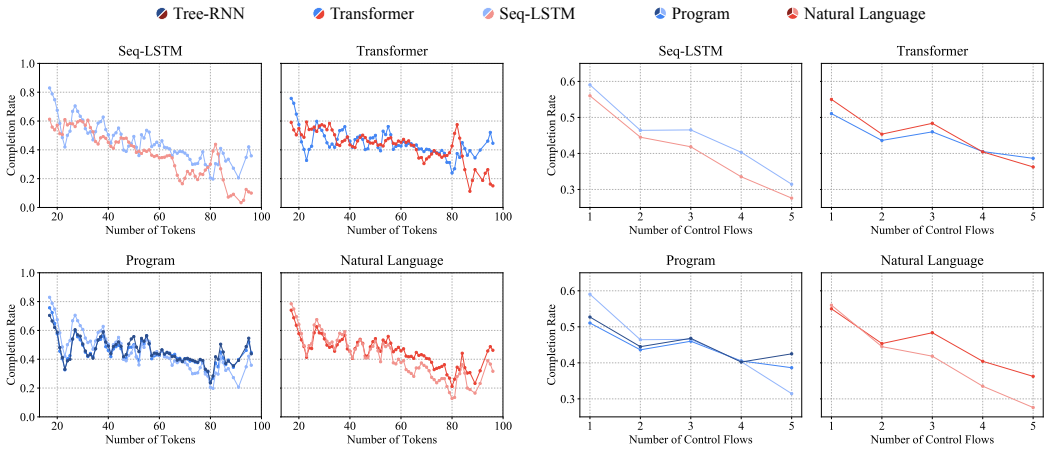
We train the proposed framework and the end-to-end learning models on training programs and evaluate their performances using the percentage of completed instructions on *test* and *test-complex* sets (shown in Table 1). Our proposed framework achieves a satisfactory *test* performance and only suffers a negligible drop (*i.e.* generalization gap) when it is evaluated on *test-complex* set. This can be attributed to the modular design, which explicitly utilizes the structure and grammar of programs, allowing the two learning modules (*i.e.* perception and policy) to focus on their local jobs. A more detailed failure analysis can be found in Section B.4.

On the other hand, all the end-to-end learning models suffer a significant performance drop from *test* to *test-complex* sets, while it is less significant for the models learning from programs, potentially indicating that models learning from instructions with explicit structures can generalize to complex instructions better. Among them, Seq-LSTM achieves the best results on *test* set, but performs the worst on the *test-complex* set. Transformer has smaller generalization gaps, which could be attributed to their multi-head attention mechanism, capturing the instruction semantics better. By leveraging the explicit structure of programs, Tree-RNN achieves the best generalization performance.

5.4.2 Analysis on end-to-end learning models

An analysis on the end-to-end learning models with respect to varying instruction length and complexity is shown in Figure 5, where all the instructions from *test* and *test-complex* sets are considered.

Instruction length. As shown in Figure 5 (a), both Seq-LSTM and Transformer suffer from a performance drop as the instruction length increases. Seq-LSTM performs better when instructions are shorter, but suffers from generalizing to longer instructions. On the other hand, Transformer may learn on a more semantic level, which leads to similar overall performances across two types of instructions. Tree-RNN leverages the structure of programs and achieves a better performance.



(a) Instruction Length

(b) Instruction Diversity

Figure 5: **Analysis on end-to-end learning models:** (a) Models learning from programs generalize better to longer instructions. Transformer is more robust to longer instructions (Upper). Tree-RNN exploiting the program structure generalizes the best, but performs worst for shorter programs (Lower). (b) Seq-LSTM learning from both instructions performs worse as the diversity increases. Transformer learns better from natural language when the instructions are less diverse (Upper). Transformer and Tree-RNN learning from programs are more consistent as the diversity increases, yet perform worse on less diverse instructions (Lower).

Instruction diversity. We define the diversity of a program based on the number of control flows it contains (*i.e.* number of branches). Figure 5 (b) shows a clear trend of performance drop of all the models. Transformer is more robust to diverse instructions which could be attributed to its better ability to learn the semantics. While Seq-LSTM learning from programs are consistently better across different levels of diversities, Tree-RNN demonstrates the most consistent performances.

5.5 Modulation mechanism for learning a multitask policy

We investigate if learning a multitask policy with the learned modulation mechanism is more effective. We compare against the two most commonly used methods: concatenating a state and goal in a raw space (*Concat raw*) or a latent space (*Concat*). An illustration is shown in Figure 4 (a). Since our state contains an environment map, which is encoded by a CNN and MLP, we experiment modulating convoluted feature maps (*Modulation conv*) or feature vectors (*Modulation fc*) or both (*Modulation conv fc*). Figure 4 (b) demonstrate that the proposed policy modulation mechanism is more sample efficient. Table 1 shows that the multitask policy learning using modulation achieves better performance on task completion.

6 Conclusion

We propose to utilize programs, structured in a formal language, as an expressive and precise way to specify tasks instead of commonly used natural language instructions. We introduce the problem of developing a framework that can comprehend a program as well as perceive and interact with the environment to accomplish the desired task. To address this problem, we devise a modular framework, *program guided agent*, which executes programs with a program interpreter by altering between querying a perception module when a branching condition is encountered and instructing a policy to fulfill subtasks. We employ a policy modulation mechanism to improve the efficiency of learning the multitask policy. The experimental results on a 2D Minecraft environment demonstrate that the proposed framework learns to reliably fulfill program instructions and generalize well to more complex instructions without additional training. We also investigate the performance of various models that learn from programs and natural language descriptions in an end-to-end fashion.

REFERENCES

Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations*, 2019.

David Andre and Stuart J Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems*, 2001.

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *National Conference on Artificial Intelligence*, 2002.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, 2017a.

Jacob Andreas, Dan Klein, and Sergey Levine. Learning with latent language. In *North American Chapter of the Association for Computational Linguistics*, 2017b.

Dzmitry Bahdanau, Felix Hill, Jan Leike, Edward Hughes, Pushmeet Kohli, and Edward Grefenstette. Learning to understand goal specifications by modelling reward. In *International Conference on Learning Representations*, 2019.

Rudy R Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations*, 2017.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

John D Co-Reyes, Abhishek Gupta, Suvansh Sanjeev, Nick Altieri, John DeNero, Pieter Abbeel, and Sergey Levine. Guiding policies with language via meta-learning. *International Conference on Learning Representations*, 2019.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, 2017a.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, 2017b.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines, 2017.

Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in Neural Information Processing Systems*, 2017.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. In *Conference on Robot Learning*, 2017.

Daniel Fried, Jacob Andreas, and Dan Klein. Unified pragmatic models for generating and following instructions. In *North American Chapter of the Association for Computational Linguistics*, 2017.

Daniel Fried, Ronghang Hu, Volkan Cirik, Anna Rohrbach, Jacob Andreas, Louis-Philippe Morency, Taylor Berg-Kirkpatrick, Kate Saenko, Dan Klein, and Trevor Darrell. Speaker-follower models for vision-and-language navigation. In *Neural Information Processing Systems*, 2018.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

Michael Janner, Karthik Narasimhan, and Regina Barzilay. Representation learning for grounded spatial reasoning. *Transactions of the Association of Computational Linguistics*, 2018.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *International Conference on Learning Representations*, 2016.

Russell Kaplan, Christopher Sauer, and Alexander Sosa. Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*, 2017.

Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *arXiv preprint arXiv:1812.02788*, 2018.

Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. Synthesizing environment-aware activities via activity sketches. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.*, 2018.

Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. Learning to describe scenes with programs. In *International Conference on Learning Representations*, 2019.

Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. Mapping instructions to actions in 3d environments with visual goal prediction. In *Empirical Methods in Natural Language Processing*, 2018.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 2016.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*, 2015.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *International Conference on Machine Learning*, 2017.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, 1998.

Deepak Pathak, Parsa Mahmoudieh, Guanghao Luo, Pulkit Agrawal, Dian Chen, Yide Shentu, Evan Shelhamer, Jitendra Malik, Alexei A Efros, and Trevor Darrell. Zero-shot visual imitation. In *International Conference on Learning Representations*, 2018.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016.

Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Neural Information Processing Systems*. 2018.

Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Advances in Neural Information Processing Systems*, 2018.

Bradly C. Stadie, Pieter Abbeel, and Ilya Sutskever. Third person imitation learning. In *International Conference on Learning Representations*, 2017.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, 2018.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *Association for Computational Linguistics*, 2015.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth J Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Association for the Advancement of Artificial Intelligence*, 2011.

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

Ziyu Wang, Josh S Merel, Scott E Reed, Nando de Freitas, Gregory Wayne, and Nicolas Heess. Robust imitation of diverse behaviors. In *Advances in Neural Information Processing Systems*, 2017.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *International Conference on Robotics and Automation*, 2018.

Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot imitation from observing humans via domain-adaptive meta-learning. In *Robotics: Science and Systems*, 2018.

A PROGRAM EXECUTION

We describe our program interpreter in Section 4.1 and provide more details in this section. The program instruction considered in this work contains the following three components: (1) subtasks, (2) perceptions, and (3) control flows. Accordingly, our *Program Interpreter* is designed to consist of (1) a parser to parse each line of the program following the grammar defined by our DSL in figure 2, and (2) a program executor which executes the program conditioning on the parsed contents. The interpreter transforms the program into a tree-like object by exploiting its structure (*i.e.* scopes) and then utilizes the parsed contents to traverse it to execute the program.

A program tree is built by representing each line of a program as a tree node. Each tree node is a data structure containing members: (1) **node.line**, the original line in the program, (2) **node.isLeaf()**, if the current node is a leaf node (*i.e.* subtask), and (3) **node.children**, all the subroutines of the current node (*i.e.* the processes under the scope of the current line of program). The interpreter will parse according to the original line contained in the node, and decide whether to call the policy if it is a leaf node (subtask) or produce a query to call the perception module, deciding which child node (subroutine) to go into.

The subroutines of a node should correspond to proper scoping of the program. For example, in Figure 1 in the main paper, the line `if is_there[River]` has subroutines `mine(Wood)`, `build_bridge()`, `if agent[Iron]<3`, and `place(Iron,1,1)`, but not `mine(Iron)`, which should be `if agent[Iron]<3`'s subroutine.

Once the program tree is built, the program executor will perform a pre-order traversal to initiate the execution. Algorithm 1 summarizes the details of the program execution utilizing the transformed program tree.

Algorithm 1 Program Execution

Require: P: program to be executed
Require: s: environmental state
Require: π: agent policy parameterized by θ, Φ: perception module
Require: node: has member node.line as the original program line and children nodes node.children

```
procedure EXECUTE(node)
    if node.isLeaf() then
        subtask = parse_subtask(node.line)
        π(subtask, θ)                                   ▷ Calls the agent policy to execute the subtask
    else
        control_flow, perception_query = parse_ctrl_percept(node.line)
        h = Φ(perception_query, s)                      ▷ Calls the perception module with a query and state
        control_flow h                                  ▷ e.g. if, while, loop, calls the subroutines depending on h
            for child in node.children do
                EXECUTE(child)
            end for
    end if
end procedure
```

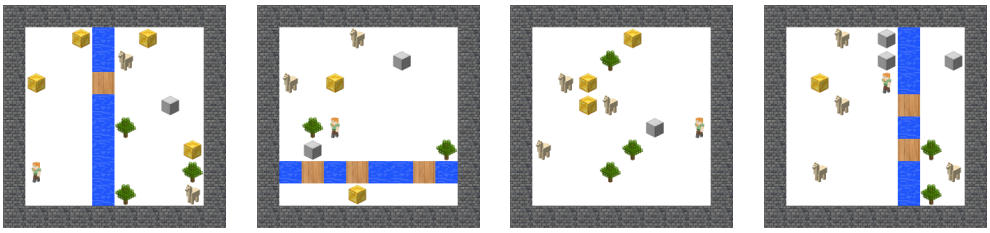
B ADDITIONAL EXPERIMENTAL DETAILS

B.1 ENVIRONMENT DETAILS

In the following paragraphs, we provide more details of the environment used in this work.

Objects in the environment. The major environmental resources that the agent can interact with are: `wood`, `gold`, and `iron`. There is a certain probability that the environment will contain a `river`, which the agent cannot go across unless a `bridge` is built (or pre-built). The environment is surrounded by brick walls, which draws the boundaries of the world.

Agent action space. The agent's actions are (1) **crafting actions**: including mining (collecting resources), placing, and building a bridge; and (2) **motor actions**: including moving to four directions



(a) (b) (c) (d)

Figure 6: Exemplar rendered environment map. The agent, objects, and stuff are represented as blocks with their corresponding textures. Specifically, the `agent` is represented as a female character. `gold` is represented as a golden block, `wood` is shown as a tree, and `iron` is represented as a sliver block. `River` is shown as a blue grid with water texture while `bridge` is presented as wooden grid. `merchant` is shown as an alpaca, which is supposed to transport the sold objects. Notice that there are 2 `merchants` in (a) and (b), while (c) and (d) contains 3 and 4 of them, respectively. The boundaries of the map are shown as brick walls.

(up, down, left, right). The crafting actions are only allowed on the current grid cell the agent is standing on, *e.g.* the subtask `mine(gold)` requires the agent to navigate to a specific location containing a `gold` with motor actions, and then perform the crafting action `mine` at the current location. To build a `bridge`, the agent should consume one of the `wood` it possesses. To sell an item, the agent needs to travel to a `merchant` and then place the item. With certain probabilities, there can be 2 to 4 `merchants` located at different locations.

Initialization. During training, when each training program is sampled, a valid environment will be randomly initialized, where validity refers to the property that the agent will be able to successfully follow the program with sufficiently provided environmental resources. At test time, we pre-sample 20 valid initialization of the environment with 20 different random seeds to ensure the validity of the two test sets.

Agent observation space (state representations). The state used in our reinforcement learning policy consists of an environment map s_{map} and an inventory status of the agent s_{inv}. s_{map} is of size $10 \times 10 \times 9$, where each channel-wise slice of size $10 \times 10 \times 1$ represents the binary existence of certain objects at a specific location, *e.g.* if $(3, 4, 2)$ is 1, it means there is a `gold` at location $(3, 4)$ (environment objects in channel dimension are zero-indexed). The objects represented by the channels are ordered as follows: `wood`, `iron`, `gold`, `agent`, `wall`, `goal` (2-D representation of intended goal coordinates), `river`, `bridge`, and `merchant`. The agent inventory status s_{inv} is augmented with the agent location, resulting in a 1-d integer vector of size 5. The ordered entry of such vector is as follows: agent's `wood` counts, agent's `iron` counts, agent's `gold` counts, agent's location coordinate x, and then y.

Goal representations. For our proposed framework, we represent the goal of the subtask as an 1-D vector of size 10, produced by the program interpreter. The first five entries of the goal vector is a one-hot representation of the subtask: `goto`, `place`, `mine`, `sell`, `build_bridge`. The 6th to 8th entries are one-hot representation of the resources: `wood`, `iron`, and `gold`. The last two entries are the intended goal locations. For example, `place(iron,3,5)` will be represented as $[0, 1, 0, 0, 0, 0, 1, 0, 3, 5]$. For end-to-end learning models, such goal representation is produced by the input encoder as a continuous latent vector representation.

Exemplar environment maps. We show several exemplary rendered environment maps in Figure 6. As can be seen, the essential resources such as `wood`, `gold`, `iron` are represented as block objects, where the `merchant` is depicted by an alpaca. The `agent` is shown as a female human character. `River` grids, with `bridge` blocks built on it is shown as the blue grid cells, where the bridge which should be transformed by wood is of wooden texture. The Boundaries can be seen in the surroundings represented as brick wall grids.

B.2 Task Instructions Details

Programs. We generate the program sets by sampling program tokens with normal distributions, and constructing them according to the DSL grammar we define. The training set is composed of on

average 32 tokens and 4.6 lines; the more complex test set, *i.e. test-complex*, contains on average 65 tokens and 9.8 lines. We include the plotted statistics of various essential properties for the three datasets in Figure 11, Figure 12, and Figure 13, respectively. Note that the maximum indent of a program is the maximum depth of its scope or the height of its transformed program tree. The number of recurring procedures includes both `while` and `loop`.

Natural language instructions. For each of the three program sets, we chunked them into several subsets of programs and assign them to annotators for their corresponding natural language translations. The annotators were instructed to read the provided DSL to understand the details of program syntax as well as some exemplary translations before they are allowed to start the task. The annotators were encouraged to give diverse and colloquial translations to avoid constantly giving dull line-by-line translations. The collected (translated) natural language instructions were then cleansed with spell checks and grammatical errors fixes. On average, the annotators used 27, 28, and 61 words to describe the instructions for the train, test, and test-complex sets respectively. The total vocabulary size of the natural language instructions is of 448.

Qualitative results on natural language analysis. We show several example data points from our testing sets in Figure 10. The leftmost column displays natural language instructions, the middle column shows our sampled ground truth programs, while the rightmost column illustrates how language can be ambiguous and lead to possible alternative interpreted programs.

B.3 Network Architectures

B.3.1 Our framework

Perception module. The perception module takes a query q and a state s as input and outputs a response h. A query has a size of 6×186, since the longest query has a length of 6 and 186 is the dimension of one-hot program tokens. Shorter queries are zero-padded to this size.

The state map s_{map} is encoded by a CNN with four layers with channel size of 32, 64, 96, and 128. Each convolutional layer has kernel size 3 and stride 2 and is followed by ReLU nonlinearity. The final feature map is flattened to a feature vector, denoted as f_m.

The state inventory s_{inv} is encoded by a two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_i.

Each token in the query is first encoded by a two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. Then, all the query token features are concatenated along the feature dimension to a single vector. This vector is then encoded by another two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_q.

All encoded features (f_m, f_i, and f_q) are then concatenated along the feature dimensions to a single vector. This vector is processed by a three-layer MLP with a channel size of 128, 64, and 32. Each fully-connected layer is followed by ReLU nonlinearity. Finally, a linear fully-connected layer produces an output with a size of 1, which should have a higher value if the response of the query is `true` and lower otherwise.

Policy. The policy takes a goal g and a state s as input and outputs an action distribution a, where the state is encoded by two types of modules: (1) a four-layer CNN encoder to encode the state map s_{map}, and (2) a two-layer MLP to encode the agent inventory status s_{inv}.

The goal g is encoded by a two-layer MLP with a channel size of 64 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_g. Given the encoded goal vector, we employ four linear fully-connected layers to predict modulation parameters $\{\gamma_i, \beta_i\}_{\{1,\ldots,4\}}$ for the state CNN encoder, where γ_1 and β_1 have size 32, γ_2 and β_2 have size 64, γ_3 and β_3 have size 96, and γ_4 and β_4 have size 128. Note that these modulation parameters are predicted for modulating convolutional features (*i.e. modulation conv*). For *modulation fc*, a linear fully-connected layer is used to produce γ^{fc} and β^{fc} with size 64.

A state map s_{map} is encoded by four-layer CNN with channel size of 32, 64, 96, and 128. Each convolutional layer has kernel size 3, strides 2, and is followed by ReLU nonlinearity. After each

convolutional layer, the produced feature maps e are modulated to $\gamma \cdot e + \beta$, where γ and β are broadcast along spatial dimensions. The final feature map is flattened to a feature vector and denoted as f_m^π.

A state inventory s_{inv} is encoded by a two-layer MLP with channel size of 64 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_i^π.

The two encoded features (f_m^π and f_i^π) are then concatenated along the feature dimension. Two fully-connected layers are used to process the feature with a channel size of 64 for both layers. Each layer is followed by ReLU nonlinearity. The final encoded feature u is then modulated to $\gamma^{fc} \cdot u + \beta^{fc}$ if *modulation fc* is used.

Finally, the modulated features $\hat{u}$ are used to produce an action distribution a and a predicted value V using two separated MLPs. Each MLP has two fully-connected layers with a channel size of 64 for both layers. A linear layer then outputs a vector with a size of 8 (the number of low-level actions). Another linear layer outputs a vector with a size of 1 as the predicted value.

B.3.2 END-TO-END LEARNING MODELS

In addition to the input encoder, the end-to-end learning models can utilize a mechanism to remember what subtasks from the instructions have been accomplished. The agent can then explicitly memorize where it stands in the instruction while completing the task. We augment such memorization mechanism utilizing the memory of another LSTM network, taking as inputs the encoded states throughout the execution trajectory. After agent taking each action, the last hidden state encoding the trajectory up to the current step is used to compute attention scores to pool the outputs of the input encoders. For Tree-RNN encoder, we simply concatenate the hidden representation from memorization LSTM with the root representation of Tree-RNN before feeding them to subsequent layers. The agent policy network then learns to perform task conditioning on this attention-pooled latent instruction vector.

We provide details of our various end-to-end learning models in Table 2. Program token embedding is jointly trained with learning the whole module, while GloVe Pennington et al. (2014) (50-D version) is used for word embedding when instructions are natural languages.

Model	Parameters	Details
Seq-LSTM	0.62M	LSTM size of 128, both program and word embeddings are of dimension 50. Attention LSTM size of 128. Attention weights of size $[256 \times 128]$, with bias of size $[128]$. Word embeddings utilize pre-trained GloVe.
Tree-RNN	0.51M	Program embeddings are of dimension 128. Attention LSTM size of 128. Composition module (to aggregate all the children representation of a node) is of size $[128 \times 128]$, and output projection weights of size $[128 \times 128]$, with bias of size $[128]$. The program embeddings are average pooled across the same program line, so that each line will be mapped to a fixed dimension representation. The composition layer is applied when combining pooled embedding from all the children of a node.
Transformer	2.63M	Number of hidden layers: 2, with 8 attention heads, and intermediate size of 256. Hidden size is 128. No dropout is applied.

Table 2: Architectural details for end-to-end learning models

B.4 FAILURE ANALYSIS

To gain a better understanding of how our proposed framework and the end-to-end learning models work or fail, we conduct detailed failure analysis on the execution traces of our model. The analysis is organized as follows:

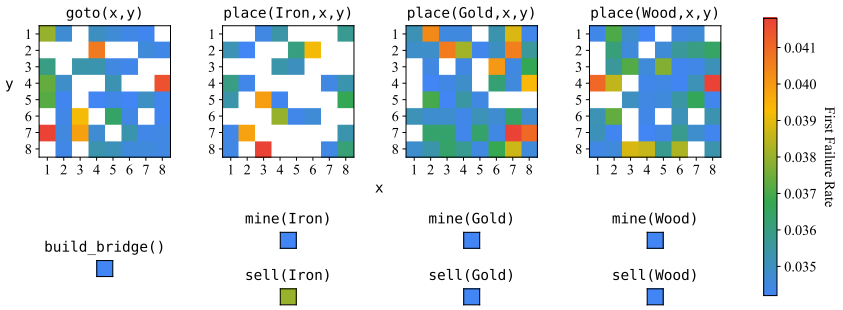


Figure 7: **First failure rate of subtasks.** Every colored grid shows the first failure rate of each subtask. From top-left to bottom-right, each block of grids show the results for subtask category `goto`, `place`, `build_bridge`, `mine`, and `sell`. Warmer colors indicate higher first failure rate; while colder colors indicate lower first failure rate. White grids indicate subtasks that either never occurs as first failed task in any execution or do not exist in the executions that lead to this figure.

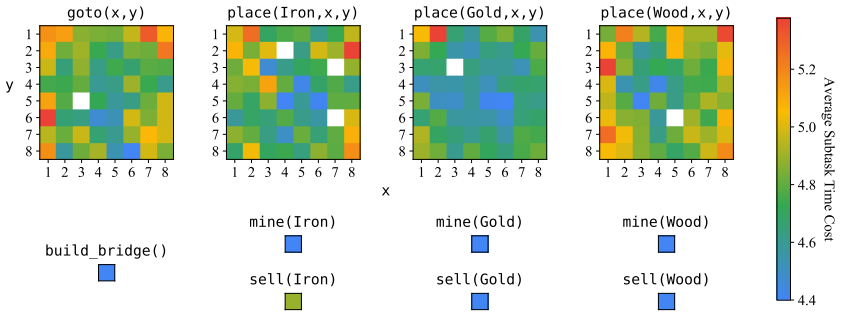


Figure 8: **Average time cost of subtasks.** The setup of this plot is similar to that of Figure 7. Warmer colors indicate higher average subtask time cost; while colder colors indicate lower average subtask time cost. White grids indicate subtasks that do not exist in the executions that lead to this figure.

- We first present an analysis of our framework on the subtasks that appear to be the first failed subtask, which immediately leads to failing the whole task. This analysis sheds some light on which subtasks most commonly cause the failure of task execution. (Section B.4.1)
- We show an analysis of how many time steps each successfully executed subtask takes on average for our framework, through which we explain which subtasks we find to be harder than others. (Section B.4.2)
- We show additional visualizations on the completion rates of different end-to-end learning models plotted with metrics not shown in the main paper, where we aim to deliver a more complete view of how these models perform. (Section B.4.3)

B.4.1 First failure rate of subtasks

As the first step of failure analysis, we want to get an idea of which subtasks cause the failure of the model in executions more often. To make this possible, we define "first failed subtask" as the first subtask that ends as a failure in an unsuccessful execution of a program. Based on this definition, we further define "first failure rate" as the percentage that an occurrence of a specific subtask turns out to be the first failed subtask of the execution that includes it.

We collect the first failure rate of all subtasks for the result we obtain from running our full model over the more complex test set, *i.e. test-complex*. The results are plotted in a visually interpretable format in Figure 7. As seen in the figure, subtasks in `goto` and `place` categories are more likely to be the first failed subtask than subtasks in `build_bridge`, `mine`, and `sell` categories. Within the `goto` and `place` subtask categories, subtasks requiring the agent to navigate to grid cells nearby the border of the world has a higher first failure rate than ones nearby the center of the world. This shows that these tasks mentioned above are more prone to failure than other subtasks.

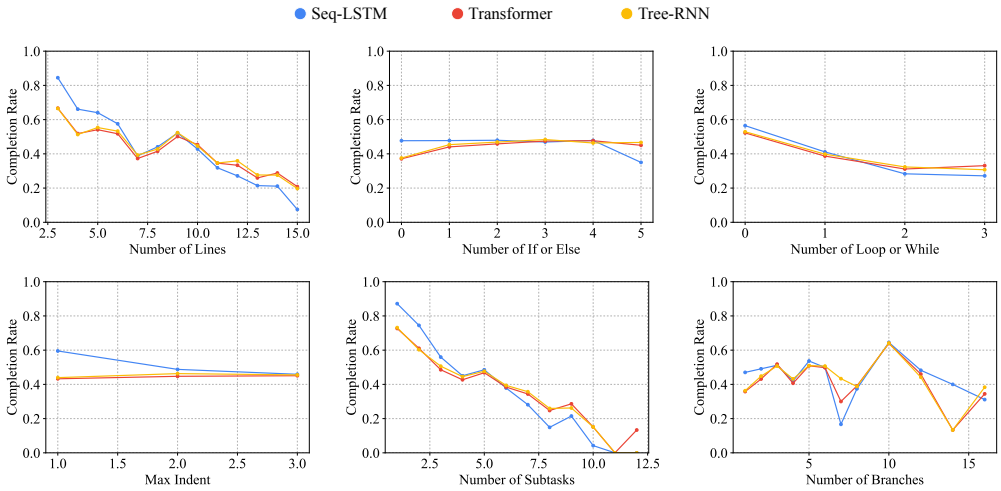


Figure 9: Additional analysis on completion rates. The results of executing program instructions on both datasets are used to produce the six plots above. In each plot, each color corresponds to a different model that we propose. For the two rightmost plots, there is a very small number of outliers that extends out of the right boundary of the plot that we omit for visual interpretability reasons. Please note that the use of colors in this figure is not the same as that in Figure 5 of the main paper.

B.4.2 AVERAGE TIME COST OF SUBTASKS

Continuing from the previous analysis, we show the average time cost of all successful subtask executions in Figure 8. As can be seen in the figure, subtasks in `build_bridge`, `mine`, and `sell` categories take relatively smaller number of time steps to complete. In `goto` and `place` categories, the closer to border the subtask requires the agent to reach, the more time consuming it gets for the agent to complete the subtask. This corresponds to the finding in the analysis of first failure rates that subtasks with destinations close to the border are more likely to fail. In other words, the closer to border the agent has to reach, the more likely it is to fail the subtask.

B.4.3 ADDITIONAL ANALYSIS ON END-TO-END LEARNING MODELS COMPLETION RATES

To conclude failure analysis, we focus on the variation of completion rates of program executions with respect to different conditioning variables. As shown in Figure 9, plots showing the trends of completion rates while evaluating with different independent variables show that execution failure is more common in cases when the program consists of larger number of lines, more loops and while statements, and larger number of subtasks. Note that this subtask count is only a summation of the occurrence counts of each subtask in a program, which does not accurately reflect the number of executions each subtask is being invoked (*i.e.* it does not reflect the repetitive counts when there is a loop).

Meanwhile, the effect of the number of if and else statements and the maximum indent values of programs on completion rates seem to vary across different models. For Seq-LSTM model, having a larger number of if and else statements or having a larger max indent value results in more failures; while for Transformer and Tree-RNN models, having larger values above results instead in fewer failures. This is probably since Transformer and Tree-RNN models are designed in a way that deals with hierarchical structures with jumps in instruction executions better (this point is also mentioned in the main paper). Despite this difference in effects, the overall change in performance when the number of if and else statements and the maximum indent value change is much less significant than that in the previous case.

During our analysis, we also designed an algorithm to calculate an estimate to the number of branches a program has. Here, the number of branches is defined by the number of distinct sets of lines that a program can be executed. For a program without control flows (no if, else-if, else, loop, and while statements), the number of branches is always 1. For if, else-if, and else statements, the exact number

#	Language Instructions	Ground Truth Program	Alternative Interpretation

(a)	If there is a river, build a bridge. Repeat the followings 3 times: mine a gold, and if environment has no more than 8 gold, mine iron, and then sell an iron.	`def run():`
		`  if is_there[River]:`
		`  build_bridge()`
		`  loop(3):`
		`  mine(Gold)`
		`  if env[Gold] <= 8:`
		`  mine(Gold)`
		`  sell(Iron)`
			`def run():`
			`  if is_there[River]:`
			`  build_bridge()`
			`  loop(3):`
			`  mine(Gold)`
			`  if env[Gold] <= 8:`
			`  mine(Gold)`
			`  sell(Iron)`

(b)	Place an iron on (7,2) and repeat 4 times, if agent has no more than 9 iron then sell a gold.	`def run():`
		`  place(Iron, 7, 2)`
		`  loop(4):`
		`  if agent[Iron] <= 9:`
		`  sell(Gold)`
			`def run():`
			`  loop(5):`
			`  place(Iron, 7, 2)`
			`  if agent[Iron] <= 9:`
			`  sell(Gold)`

(c)	Mine wood first. If agent has more than 3 iron, mine wood. If there is gold in the environment, place iron at (3,7).	`def run():`
		`  mine(Wood)`
		`  if agent[Iron] >= 4:`
		`  mine(Wood)`
		`  if is_there[Gold]:`
		`  place(Iron, 3, 7)`
			`def run():`
			`  mine(Wood)`
			`  if agent[Iron] >= 4:`
			`  mine(Wood)`
			`  if is_there[Gold]:`
			`  place(Iron, 3, 7)`

Figure 10: Exemplar data and languages ambiguity. The goal of the examples above is to show that natural language instructions while being flexible enough to capture the high-level semantics of the task, can be ambiguous in different ways and thus might lead to impaired performance. In example (a), the modifier "repeat the following 3 times" has an unclear scope, resulting in two possible interpretations shown in program format on the right side; in example (b), "repeat 4 times" can be used to modify either the previous part of the description or the latter part of it, resulting in ambiguity; in example (c), the last sentence starting with "If" has unclear scope. In all of the above cases, a model that learns to execute instructions presented in natural language format might fail to execute the instructions successfully because of the ambiguity of the language instructions.

of branches these statements incur can be calculated easily. In cases of the loop and while statements, we treat loops as being executed only once and while statements as if statements when we calculate the number of branches. The result shown in the analysis does not reveal a clear trend. We attribute this result to two possibilities – either the metric we create is not accurate enough, or it is not a very suitable metric to be inspected.

B.5 HYPERPARAMETERS

We use the following hyperparameters to train A2C agents for our model and all the end-to-end learning models: learning rate: 1×10^{-3}, number of environment: 64, number of workers: 64, and number of update roll-out steps: 5.

B.6 COMPUTATIONAL RESOURCES

We train all our models on a single Nvidia Titan-X GPU, in a 40 core Ubuntu 16.04 Linux server.

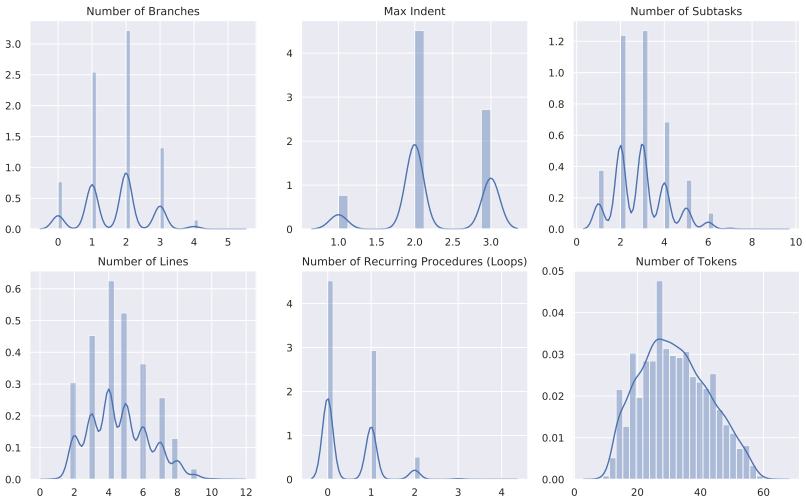


Figure 11: Program set statistics for training set (*train*).

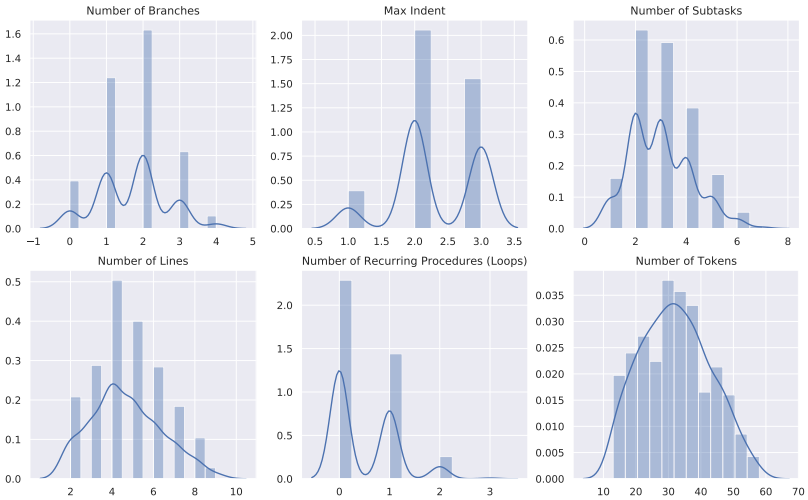


Figure 12: Program set statistics for same complexity testing set (*test*).

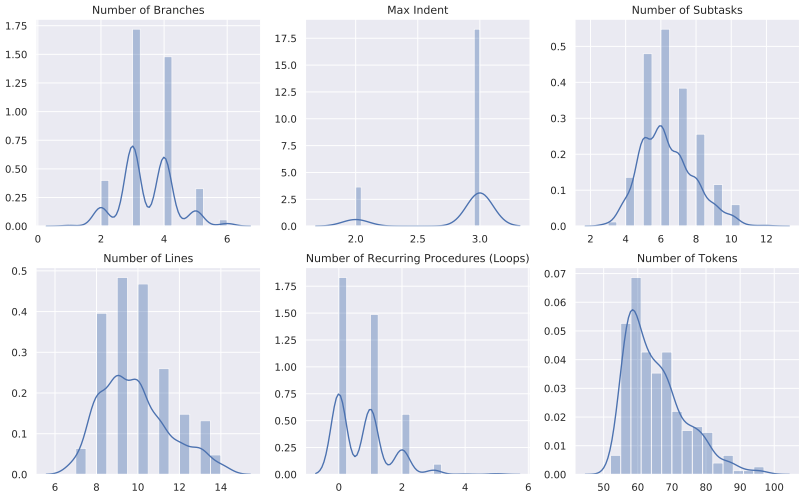


Figure 13: Program set statistics for more complex testing set (*test-complex*).