Hierarchical Control of Situated Agents through Natural Language

Shuyan Zhou, Pengcheng Yin, Graham Neubig
Language Technologies Institute
Carnegie Mellon University
{shuyanzh, pcyin, gneubig}@cs.cmu.edu

Abstract

When humans perform a particular task, they do so hierarchically: splitting higher-level tasks into smaller sub-tasks. However, most works on natural language (NL) command of situated agents have treated the procedures to be executed as flat sequences of simple actions, or any hierarchies of procedures have been shallow at best. In this paper, we propose a formalism of *procedures as programs*, a method for representing hierarchical procedural knowledge for agent command and control aimed at enabling easy application to various scenarios. We further propose a modeling paradigm of *hierarchical modular networks*, which consist of a *planner* and *reactors* that convert NL intents to predictions of executable programs and probe the environment for information necessary to complete the program execution. We instantiate this framework on the IQA and ALFRED datasets for NL instruction following. Our model outperforms reactive baselines by a large margin on both datasets. We also demonstrate that our framework is more data-efficient, and that it allows for fast iterative development.[1]

1 Introduction

Procedural knowledge, or "how-to" knowledge, refers to knowledge of how to execute particular tasks. It is inherently hierarchical; high-level procedures consist of many lower-level procedures. For example, "cooking a pizza" comprises many lower-level procedures, including "buying ingredients", "knead dough", etc. There are also multiple levels of hierarchy; "buying ingredients" can be further decomposed to "go to a grocery", "paying" etc.

There has been significant prior work on benchmarks and methods for complex task completion using situated agents given natural language (NL) instructions, such as agents trained to navigate the web and mobile UIs (Li et al., 2020; Xu et al., 2021)

---

[1]All code will be released upon acceptance.

or solve household tasks (Shridhar et al., 2020a). However, most methods applied to these tasks use a *reactive* strategy that makes decisions on the low-level atomic actions available to the agent while making steps through the environment (Gupta et al., 2017; Zhu et al., 2020), or define procedures in a shallow way where there only exists one level of hierarchy (Andreas et al., 2017; Gordon et al., 2018; Yu et al., 2019; Das et al., 2019).These approaches are often data-inefficient due to the semantic gap between abstract natural language instructions and concrete executions. In contrast, several works have demonstrated that using specially designed intermediate representations tailored to individual tasks (Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013; Misra et al., 2016) can help reduce this expense and improve performance, albeit at the cost of significant effort on the part of the researchers devising these methods.

In this paper, we propose a framework to improve the execution of complex natural language commands (example in Fig. 1) by expressing *procedures as programs* (PaP) written in a high-level programming language like Python (§4). This makes it easy for human engineers to *express and leverage their hierarchical procedural knowledge*, and the execution of each program yields actions to accomplish a task described in NL. There are several merits to this approach. First, programs are inherently hierarchical; they apply nested function calls to realize higher-level functionality with multiple calls to lower-level functionality. Second, programs have built-in control-flow operators, making it possible to deal with multiple divergent situations without the loss of higher-level abstraction. Third, programs provide a flexible way to define, share and call different machine-learned components to perceive the environment through an embodied agent's executions. Finally, programs in a familiar high-level programming language are comprehensible and curatable, allowing for fast development on
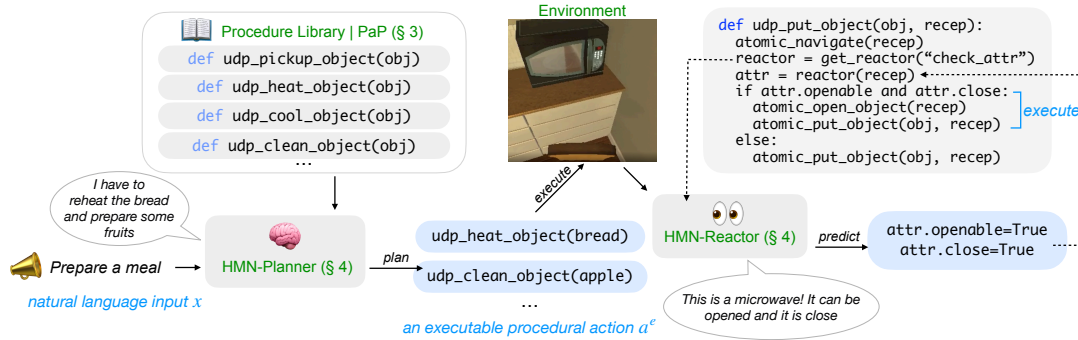
Figure 1: The proposed framework, containing a hierarchical library of procedures written as Python functions (§4). Coupled with this library is a hierarchical neural network (HMN, §5) with a PLANNER that constructs an executable procedure and REACTORS that react to the environment to resolve control flow.

various tasks. These four features remain largely unexplored in the existing representations (Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013; Misra et al., 2016), as discussed further in §2.

Coupled with this representation, we propose a modeling paradigm of *hierarchical modular networks* (HMN; §5) that has (1) a learnable PLAN-NER that maps NL to the corresponding executable programs and (2) a collection of REACTORS that perceive the environment and provide context-sensitive feedback to decide the further execution of the program. Such modular design can facilitate training efficiency and improve the performance of each individual component (Andreas et al., 2016).

We instantiate our framework on two task settings: the IQA dataset (Gordon et al., 2018) where an agent explores the environment to answer questions regarding objects; and the ALFRED dataset (Shridhar et al., 2020a), in which an agent must map natural language instructions to actions to complete household tasks (§6). In experiments (§7), we find that our framework outperforms the reactive baseline by a significant margin on both datasets, and is significantly more data-efficient. We also demonstrate the flexibility of our framework for fast iterative development of program libraries. We end with a discussion of the limitations of the framework and the potential solutions, paving the way for future works that scale our framework to more open-domain tasks (§7).

2 Contrast to Previous Formalisms

While designing intermediate representations that stand between NL and low-level actions *for individual tasks* has been studied in the literature, our goal is to design a framework that makes it simple to design such representations *for new tasks*, with a

particular focus on capturing the hierarchical nature of procedures. In contrast to most previous works in this area, which employ relatively esoteric representation methods such as lambda calculus (Artzi and Zettlemoyer, 2013; Artzi et al., 2014), PaP uses widely-adopted general-purpose programming languages (*e.g.* Python) to specify and represent hierarchical procedures. These are comprehensible to most engineers and do not require system designers to learn a new task-specific language. PaP also enable easy creation of more hierarchical procedures with *reusable sub-routines*. Existing works either do not model such sub-procedures as reusable components (Misra et al., 2016), or define procedures as a flat sequence of actions without any hierarchy (Chen et al., 2020; Artzi and Zettlemoyer, 2013). The hierarchical procedures with reusable sub-routines is also reminiscent of works in semantic parsing, which compose programs from idiomatic program structures (Iyer et al., 2017; Shin et al., 2019). More discussions are in §E.

Additionally, PaP uses control flow with divergent branches to handle environment-specific variations of a high-level procedure. A single procedure could therefore dynamically adapt to a variety of environments following the branches triggered by the environments. This makes our representations more compact. This feature also allows developers to easily inject human priors of executions traces under different conditions, which might be challenging to learn in a data-efficient manner. To our best knowledge, this feature is largely unexplored in the literature on designing intermediate representations for agent control.

Finally, PaP provides a convenient interface for procedures to query and interact with task-specific situated components (*e.g.* a visual component). Un-

der PaP, situated components are exposed as pre-defined APIs, and can be easily called by high-level procedures. In contrast, existing works either require separate mechanisms to call such components (Misra et al., 2016), or the environment where they are expected to work is less complex, and thus the flexible use of a collection of situated components is not a necessity (Chen and Mooney, 2011).

We can also view the PaP formalism as a way to construct behavior trees (Colledanchise and Ögren, 2018), which have been used in robotic planning and game design literature. We can use the off-the-shelf tools to convert the programs to abstract syntax trees (AST) which resemble these trees. Previous works on robotics also leverage planning domain definition language (PDDL) and answer set planners (ASP) for task planning (Jiang et al., 2019b), which is conceptually different from our formalism. PDDL+ASP searches for an action sequences based on the initial and the final states, while our formalism focuses on describing the actual procedure used to accomplish a task.

3 Task: Controlling Situated Agents

First, we define the task of controlling an agent in some situated environment $\mathcal{E}$ through natural language. The environment $\mathcal{E}$ provides a set of atomic actions $\mathcal{A}^a = \{a_1^a, a_2^a, ...\}$ to interact with the environment. Each atomic action can take zero or more arguments that specify which parts of the environment to which it is to be applied. We denote action a_i^a's jth argument as $r_{i,j}$. The specific type of each argument will depend on the action and environment; it could be discrete symbols, scalar values, tensors describing regions of the visual space, etc. Given a user intent x, the control system aims at creating an atomic action sequence consisting of a sequence of actions $\boldsymbol{a} = [a_1, a_2, ...]$ ($a_i \in \mathcal{A}^a$) and concrete assignments $\boldsymbol{r}$ for each of these n actions. This action sequence is executed against the environment to achieve a result $\hat{y} = \mathcal{E}(\boldsymbol{a}, \boldsymbol{r})$, which is compared against a gold-standard result y using a score function $s(y, \hat{y})$. Action sequences realizing the intent will receive a high score, and those that do not will receive a low score.

4 Representing Procedures as Programs

Next, we introduce the main components of our formalism. A few examples are listed in Tab. 1.[2]

---

[2]Since actions are implemented as functions, we use "action" and "function" interchangeably.

```python
# C1: an atomic action to toggle on an appliance
def atomic_toggle_on(obj):
    env.call("toggle_on", obj)
# C2: a procedural action to pick and then put an object
def udp_pick_and_put_object(obj, dst):
    udp_pickup_object(obj)
    udp_put_object(obj, dst)
# C3: an emptying receptacle procedure with for-loop
def udp_empty_recep(recep, dst):
    reactor = get_reactor("find_all_obj")
    obj_list = reactor(recep)
    for obj in obj_list:
        udp_pick_and_put_object(obj, dst)
# C4: a pickup object procedure with control flow
def udp_pickup_object(obj):
    atomic_navigate(obj)
    reactor1 = get_reactor("find_recep")
    reactor2 = get_reactor("check_obj_attr")
    recep = reactor1(obj)
    attr = reactor2(recep)
    if attr.openable and attr.close:
        atomic_open_object(recep)
        atomic_pickup_object(obj)
        atomic_close_object(recep)
    else: atomic_pickup_object(obj)
```

Table 1: Atomic and procedural action functions in Python, starting with `atomic` and udp respectively.

Interface to Atomic Actions $\mathcal{A}^a$ (C1) Atomic actions provide a medium for direct interaction with the environment. The call of an atomic action with proper argument types will invoke the corresponding execution in the environment.

Procedural Actions $\mathcal{A}^p$ (C2-C4) Procedural actions describe abstractions of higher-level procedures composed of either lower-level procedures or atomic actions. Notably, lower-level procedures can be *re-used* across many higher-level procedures without re-definition. Formalizing the hierarchies in this compact way can not only facilitate the procedure library curation process but also potentially benefit automatic library induction (e.g. through minimal description length (Ellis et al., 2020)).

Control-flow of $\mathcal{A}^p$ (C3-C4) There can be multiple execution traces to accomplish the same goal under different conditions. For example, picking up an object from inside a closed receptacle requires opening the receptacle first, while the open action is not required for objects not in a receptacle. To improve the *coverage* of procedural functions we leverage the built-in control flow of the host programming language to allow for conditional execution of environment-specific actions (**C4**). To deal with the repeated calls of the same routine, we further introduce for/while-loops. For example, **C3** works for emptying receptacles with variable number of objects without repeatedly writing down the `udp_pick_put_object`. Leveraging control flows to describe divergent procedural traces remains largely unexplored in previous works.

Call of Situated Components (C3-C4) The dynamic trigger of a control flow often remain unknown before the agent interacts with the environment. We introduce situated components to probe the environment and gather state information to guide program execution. In **C4**, the agent uses two different reactors to find the potential holder of an object (`reactor1`) and exam the holder's properties (`reactor2`). A reactor can be implemented in many ways (*e.g.* using a neural network).

5 Hierarchical Modular Networks

This section introduces how to use the procedure library $\mathcal{A}$ to generate executable programs to complete tasks described in natural language x. We propose a modeling method of *hierarchical modular networks* (HMN) that consists of two main components. First, there is a HMN-PLANNER that convert x to an executable procedural action $a^e = \{a_1, a_2, ..., a_n\}$ where a_i either belongs to atomic functions $\mathcal{A}^a$ or procedural functions $\mathcal{A}^p$. We model the HMN-PLANNER as a sequence-to-sequence model where the encoder takes x as input, and the decoder generates one function a_i at a time from a constrained vocabulary $\mathcal{A}^p \bigcup \mathcal{A}^a$, conditioned on x and the action history $\{a_1, ..., a_{i-1}\}$.

Next, we define the collection of situated components, "reactors," as HMN-REACTORS. Each reactor is a classifier that predicts one or many labels given the observed information (*e.g.* the NL input, the visual observation. For example, `reactor2` in **C4** in Tab. 1 probes the status of a receptacle based on receptacle name and the visual input. HMN-REACTORS allows us to flexibly share the same reactor among different functions and design separated reactors to serve different purposes. For example in **C4**, we use two reactors to find the possible receptacle of an object (`reactor1`) and to perceive the open/closed status of a receptacle (`reactor2`) since these two tasks presumably require more mutually exclusive information. At the same time, we share `reactor2` to also probe the related `openable` property of a receptacle for more efficient parameter sharing. This sort of modular design leads to efficient training and improved performance (Andreas et al., 2016).

6 Instantiations

In this section, we introduce two concrete realizations of the proposed framework over the IQA dataset (Gordon et al., 2018) and the ALFRED dataset (Shridhar et al., 2020a). Both are based on egocentric vision in a high-fidelity simulated environment THOR (Deitke et al., 2020).

6.1 IQA

IQA is a dataset for situated question answering with three types of questions querying (1) the existence of an object (*e.g. Is there a mug?*), (2) the count of an object (*e.g. How many mugs are there?*) and (3) whether a receptacle contains an object (*e.g. Is there a mug in the fridge?*).

There are seven atomic actions in IQA, *i.e.* `Moveahead`, `RotateLeft`, `RotateRight`, `LookDown`, `LookUp`, `Open` and `Close`; and all arguments are expressed through the unique object IDs (*e.g.* `apple_1`). We further process the atomic navigation actions to a single atomic action `Navigate` with one argument `destination`, which moves the agent directly to the destination. This replacement is done by searching the scene and recording the coordinates of unmovable objects (*e.g.* cabinet) – more details provided in the §C.1.

Procedure Library We design a procedure for each of the three types of questions in IQA, as shown in Tab. 2. Generally speaking, those procedures first search all or a subset of the receptacles (*e.g.* `table`, `fridge`) in a scene for the target object (*e.g.* `mug`), and then execute a question-specific intent (*e.g.* existence-checking, counting). Tab. 2 shows the procedure for answering existence questions. Since the target object can be inside a receptacle (*e.g.* fridge), we introduce control flow to decide whether to open and close a receptacle before and after checking its contents in sub-procedure `udp_check_relation`. Following the paper author's understanding of the three types of questions, these procedural functions were created without looking into any actual trajectories that answer these questions. In total, we define six procedural actions with a complete list in §A.

HMN The natural language questions x in IQA are generated with a limited number of templates. There are only seven receptacles, and three of them are openable. We thus use a rule-based HMN-PLANNER to map a template to one of the three high-level procedural actions (*i.e.* existence, count and contain). Then, we design two reactors, each as a multi-classes classifier: ATTRCHECKER, which examines the properties (whether the object is openable) and the status (whether the object is opened) of an object, and RELCHECKER, which checks the spatial relation between two objects. We leave the

```python
# check existence of an object in the scene
def udp_check_obj_exist(obj):
    all_recep = udp_grid_search_recep()
    for recep in all_recep:
        rel = udp_check_relation(obj, recep)
        if rel == OBJ_IN_RECEP:
            return True
    return False
# check object inside receptacle
def udp_check_relation(obj, recep):
    atomic_navigate(recep)
    r1 = get_reactor("check_obj_attr")
    r2 = get_reactor("check_obj_recep_rel")
    attr = r1(recep)
    if attr.is_openable and attr.is_closed:
        atomic_open_object(recep)
        rel = r2(obj, recep)
        atomic_close_object(recep)
    else:
        rel = r2(obj, recep)
    return rel
```

Table 2: The procedural actions to answer the existence questions of the IQA dataset.

detailed implementations of the reactors to §C.3. Notably, we use *zero* IQA training data to build the HMN. Instead, it is made up of a few heuristic components based on the predictions of a pre-trained perception component.

6.2 ALFRED

ALFRED is a benchmark for mapping NL instructions to actions to accomplish household tasks in the situated environment (*e.g.* heat an egg). Examples in ALFRED come with both single-sentence high-level intents describing a goal (*e.g.* the NL input in Fig. 1), and more fine-grained, step-by-step instructions. In this paper we only use the high-level intents, a more realistic yet more challenging setting to study the effectiveness of our framework in encoding extra procedural knowledge for under-specified intents. Besides the seven atomic actions in the IQA dataset, ALFRED also introduces `Pickup`, `Put`, `ToggleOn`, `ToggleOff` for object interactions. ALFRED uses 2D binary tensor describing regions of the visual space as arguments. Similarly to IQA, we replace the navigation action with an atomic action `Navigate destination`. Previous works also apply similar replacement (Shridhar et al., 2020b; Karamcheti et al., 2020) to allow the agent to proceed to a location without fail. Details in §C.

Procedure Library We create a procedure library for ALFRED by identifying idiomatic control flow and operations from a small set of randomly sampled examples. The library is designed with two goals in mind as discussed in §4: *reusability*, where a single function can be applied to multiple similar scenarios, and *coverage*, where a function

should cover different execution trajectories under different conditions For instance, many tasks consist of a sub-routine to obtain an object by first navigating to the object and then picking up the object by hand, calling for a reusable procedure adaptable to those scenarios. Moreover, if an object is positioned inside a receptacle, picking up the object would require opening the receptacle first, an edge case that should be covered by relevant procedures (*e.g.* **C4** in Tab. 1). Notably, we constrain the conditions of the control flow to the logic operation of the property values of objects (*e.g.* `fridge.is_openable=True`).

In total, we define ten such procedural actions (complete list in §A). This creation process was done by the first author, a graduate student proficient in Python, and took about two hours. This modest amount of time is partially due to PaP's intuitive interface that allows for quick summarization of complex procedures and partially due to ALFRED's relative simplicity; it has a limited number of task types and consistent execution traces. A sanity check of an initial version of the library uncovered some mismatches (details in §C.4). For example, a laptop should be closed before picking up, which was not captured by our library. We thus added a `udp_close_if_needed` function call before the `atomic_pick_object` in `udp_pick_object`. On one hand this increases the complexity of the library design process, but on the other hand it also demonstrates the flexibility of the PaP framework, as the necessary fixes could be done entirely by modifying the procedure library itself. §7.1 provides an end-to-end comparison with different procedural libraries.

To investigate the scalability of our annotation process, we also provided a similar guideline and the 21 examples to a separate programmer who does not have any prior knowledge to the dataset. We found that the programmer could quickly understand the PaP Python interface and issue reasonable procedural functions that highly resemble our own creations. This indicates the possibility to curate the procedure libraries with crowd-sourcing efforts. More discussion is provided in §7.2 and the full list of the annotation guideline and the user-issued functions are listed in §B.

HMN As discussed in §5, HMN-PLANNER generates an executable procedural action a^e, given the natural language instruction x. We implement our planner with a sequence-to-sequence model with

71

```python
# C1, heat an object with microwave
def udp_heat_object(obj):
    udp_pick_and_put_object(obj, microwave)
    atomic_toggleon_object(microwave)
    atomic_toggleoff_object(microwave)
# C2, prepare the receptacle for future interactions
def udp_prepare_recep(obj):
    reactor = get_reactor("check_obj_attr")
    attr = reactor(obj)
    if attr.is_openable and attr.is_closed:
        atomic_open_object(obj)
```

Table 3: Two procedural actions for ALFRED

attention (Bahdanau et al., 2015).

Based on the construction of the procedure library and the required argument type, we design three reactors: ATTRCHECKER, which has the same functionality as in IQA, REFINDER, which probes where the desired object lies by predicting a receptacle name from all available receptacles to the dataset, and MGENERATOR, which generates the 2D binary tensor representing the interaction region. Since ALFRED has much richer scene configurations and more diverse objects than IQA, the reactors are fully implemented with neural networks. This demonstrates the flexibility of our framework to share, add and replace components to suit different situations. We describe the detailed implementations of the reactors in §C.3. The HMN is trained in a supervised fashion, and the heuristic way to induce the supervisions from the original dataset is described in §C.4.

7 Experiments

We compare our proposed framework with the baseline reactive agents that predicts a single atomic action at each time step. Notably, we apply the same pretrained vision models, pre-searched map and the `Navigate` atomic action used in PaP-HMN to the baseline to ensure a fair comparison. More details in §C.2. We attempt to answer the following research questions: (1) Does our framework performs better in complex tasks with inherent hierarchical structures, comparing to a purely reactive system? If so, in what way? (2) Can our framework leverage the procedural knowledge encoded in the procedure library and the modularity of its HMN to learn more efficiently? And (3) Can our framework accelerate the development of the task of interest?

7.1 Results on IQA

Results in Tab. 6 show that our framework yields the best performance across all models over different question types. Through error analysis, we

---

³*unseen* features the out-of-distribution visual appearances and arrangements of objects, same for ALFRED

		EX	CNT	CT
A3C	*seen*	-	-	-
	unseen	48.6	24.5	49.9
HIMN	*seen*	73.7	36.3	60.7
	unseen	68.5	30.4	58.7
Reactive	*seen*	50.0	25.1	49.6
	unseen	18.9	9.1	30.6
PaP-HMN	*seen*	**82.8**	**43.8**	**82.2**
	unseen	**83.8**	**45.2**	**83.1**
PaP$_{v0.1}$-HMN	*seen*	80.3	41.5	75.7

Table 4: The answer accuracy (%) over IQA dataset on existence (EX), counting (CNT) and contain (CT) questions. The results of AC3 and HIMN are from Gordon et al. (2018). **Bold** shows the best performance[3]

observe that while the reactive model can generate reasonable action sequences *seen*, its answers are no better than a random guess. This indicates the inability of a reactive model to book-keep the observed objects in the memory. For *unseen*, we find that the baseline model skips predicting some receptacles or even generates syntactically invalid sequences (*e.g.* functions without required arguments). This is surprising, since the reactive baseline is trained using the *canonicalized* action sequences according to the roll-out of the for-loops in the procedure library, which are quite regular. This indicates that even simple repeated procedures can be easily represented with a for/while-loop can still be challenging to a reactive agent implemented with a sequence-based backbone. The strong performance of PaP might seem unsurprising given that the library is tailored carefully to the domain. However, sophisticated models like HIMN (Gordon et al., 2018) still struggle to capture such simple patterns, and there is not a straightforward way to plug the simple rules that we were easily able to describe in PaP in to improve its performance; PaP solves the easy problems so that an ML model can focus its effort on the more challenging problems that truly require learning (*e.g.* object grounding).

Procedure Library Manipulation One advantage of our approach is that it decouples the reactors from the creation of the procedural knowledge, thus allowing plug-in update of the procedure library without time-consuming redesigning or retraining the reactors. Tab. 5 lists two versions of the procedure that decides the list of receptacles to enumerate, and the results of v0.1 are shown at the bottom of Tab. 4. In v0.1, the agent stands in its randomly initialized position, looks around, and detects receptacles. Only the detected recepta-

```
# v0.1: only scan at the start position
def udp_search_recep():
    r = get_reactor("detect_recep")
    receps = []
    for rotation in range(0, 360, 90):
        atomic_rotate(rotation)
        for horizon in [-30, 0, 30]:
            atomic_look(horizon)
            receps += r()
    return receps
# now: navigate to every reachable point and scan
def udp_grid_search_recep():
    if not done_search:
        all_receps = [] # global var
        for pos in reachable_pos:
            atomic_navigate_pos(pos)
            all_receps += udp_search_recep()
    return all_receps
```

Table 5: Two versions for getting receptacles.

	seen	*unseen*
Singh et al. (2020)	5.4	0.2
Reactive	21.0	5.6
PaP-HMN	**27.0**	**11.7**
Reactive + Oracle MG	40.7 (48.6)	36.4 (45.0)
PaP-HMN + Oracle MG	**54.5 (61.0)**	**51.3 (61.1)**

Table 6: The full task success rate SR (the partial task success rate, SSR, %) of the baseline reactive model and our model. MG represents the mask generator. **bold** shows the best performance for each setting.

cles are checked to answer the question. However, since not all receptacles are visible to the agent at the agent's initial point, such checking could be incomplete. We upgraded this function to the new version where the agent searches all possible positions of the scene and memorizes the unmovable receptacle positions. This process only happens once for a scene, and the searched map is stored for future uses. In this way, most receptacles are covered. This simple modification without changing the remaining parts of the framework improved the CT answer accuracy by 6.6% and improvement of around 2.5% over the other two question types.

7.2 Results on ALFRED

Tab. 6 lists the results. Our model yields a consistent gain over the baseline system on both splits.[4] In our analysis, we find that the Mask R-CNN vision model is the main bottleneck of both end-to-end systems, which we hypothesis is due to the sub-optimal transfer from the MSCOCO (Lin et al., 2014) to the ALFRED data. It frequently misclassifies the object types or does not recognize the object in the scene at all. This results in the failure of object grounding and thus the failure of the task

---

[4]Singh et al. (2020) predicts atomic navigation sequences (*e.g.*MoveAhead) instead of Navigate. The agent struggles to navigate to destination with only high-level goal. This shows the difficulty of navigation under our experiment setting.

completion. Since the development of a better object detector is somewhat orthogonal to our main contributions, to isolate the impact of using a weak object detector on the end-to-end performance, we replace the Mask R-CNN with an oracle object mask generator, which always localize and interact with the provided object name if the object is in view for all experiments below. We observe a larger performance gap using this oracle mask generator as shown in the bottom half of Tab. 6. This gap suggests that procedural knowledge that could be summarized as several functions describable within a short period of time (in this case, ten functions in two hours) can still be difficult for a reactive system to capture. While the same procedural knowledge can be used in many cases with different environment dynamics, a reactive system struggle to distill such knowledge when interacting with highly diverse and dynamic environments.

Performance w.r.t. Action Length In Fig. 2, we further break down the results to buckets w.r.t the length of atomic action sequences (without arguments), which roughly represents the difficulty of a task. We observe consistent improvements over all buckets, This difference is even more obvious for challenging tasks with over 21 atomic actions. Our model maintains similar performance for such cases on *seen*, and being able to accomplish 30% tasks successfully on *unseen*, while the baseline can barely complete any task. These suggest our framework's stronger capacity to solve long-horizon tasks of deeper hierarchies.

Data Efficiency The hierarchical procedural knowledge could potentially allow the system to learn task completion in a data-efficient manner. We benchmark HMN with varying amounts of training data. As shown in Fig. 2, with 20% of the training data, our method exceeds the baseline with the full training set by a large margin (7.7% and 17.3% respectively). Furthermore, for *seen*, the baseline only obtains less than 60% SR with 20% training data, compared to the full data; our method could maintain around 90% SR of the full data setting. These strongly demonstrate the data efficiency of our method.

Few-shot Generalization Next, we test if our framework can generalize to novel compositional procedures with relatively supervised examples. We design the few-shot experiments where a subset of the executable procedural actions (a^e) are held out, and we sample at most 20 samples of each
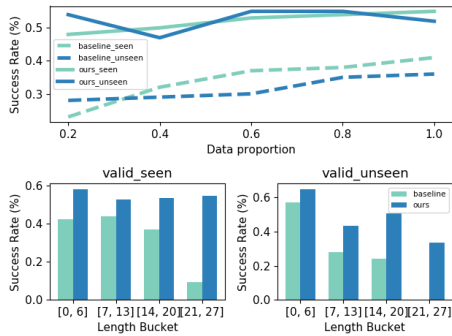
Figure 2: The SR (%) with proportions of the full training set (top) and on each length bucket of the *seen, unseen* (bottom).

$\boldsymbol{a}^e$ and add them to the training set. We evaluate the model on these held-out $\boldsymbol{a}^e$. We use two strategies to choose the held-out set; the first randomly selects n $\boldsymbol{a}^e$; the other selects the longest n $\boldsymbol{a}^e$ ($n = 4/19$). PaP-HMN achieves **33.1** and **44.9** SR with these two strategies while the reactive baseline only reaches 13.9 and 3.3 respectively[5]. Our method consistently outperforms the baseline by a large margin on both settings, which strongly demonstrates our method's generalization ability in the few-shot scenario. The significant gain under the short to long setting shows our method's strong capacities in completing long-horizon tasks in a data-efficient way compared to the baseline.

Analysis Our framework brings several advantages. First, compared to low-level actions, the high-level procedural functions are better aligned with abstract NL inputs. This thus benefits the learning and the prediction of PLANNER. Second, programs maintain the consistency of the actions, while a reactive agent might make inconsistent predictions, especially arguments, between actions. Finally, the modular design of PLANNER and the RE-ACTORS improve the robust behavior of the agent. More discussion with examples is in §D.1.

Next, we investigate failure cases. First, our ablation study shows that PLANNER correctly predicts 80% of executable procedural actions $\boldsymbol{a}^e$, and the failures are mainly due to rare words (*e.g. soak a plate*). In addition, we manually annotated 50 failed examples whose $\boldsymbol{a}^e$ are correct. We found that 26 failures are due to the sub-optimal interaction positions of the receptacles that we compute during the pre-search phase (§C.1). This causes the interaction with a visible object or receptacle to fail. The pre-search map also missed some objects, and navigating to these objects always failed. Besides,

```
def udp_heat_object(obj):
    reactor = get_reactor("find_qualified_appliance")
    app = reactor(obj) # (e.g. microwave, oven)
    udp_navigation(app)
    atomic_reactor = get_reactor("predict_atomic_action")
    atomic_action = atomic_reactor(app)
    while atomic_action != STOP:
        env.call(atomic_action)
        atomic_action = atomic_reactor(app)
```

Table 7: A potential rewriting of **C1** of Tab. 3.

the REACTOR prediction errors fail on 18 examples; ambiguous annotations caused two errors, and the wrong argument prediction of the PLANNER caused four errors. §D.2 shows a comprehensive discussion with potential solutions.

8 Limitations and Future Work

Overall, our experiments demonstrate the benefit of our framework for encoding hierarchical procedural knowledge, especially under low-data or few-shot generalization regimes. One limitation of the experiments here is that they covered domains where it is relatively easy to enumerate the tasks that must be solved in the domain. One intuitive solution in situations where this is not possible is to manually create libraries that cover major procedures but fall back to atomic/reactive control when necessary. For example, as in Tab. 7, the program can call a reactor implemented as a the neural network (`atomic_reactor`) to predict atomic actions when using different appliance to heat an object, instead of enumerating different conditional branches. Another possibility is to automate procedure library creation through mining structured procedural knowledge from Web (Tenorth et al., 2010; Kunze et al., 2010), or through induction of high-level procedures from corpora of atomic action sequences (Ellis et al., 2020).

Another interesting note is that though hierarchical procedural knowledge is ubiquitous in human daily life, most existing NL instruction following benchmarks do not feature such complex, hierarchical procedures. Although there can be hierarchies embedded in vision-language navigation tasks (Anderson et al., 2018), game playing through reading documentation (Zhong et al., 2019) or through NL communication (Suhr et al., 2019; Jernite et al., 2019) and mobile phone navigation (Li et al., 2020), the hierarchies are shallow at best, or the occasional complex ones are limited in their breadth. Therefore, creating NL instruction following benchmarks that feature more realistic and diverse procedures is one final important direction for future work.

---

[5]For *random split*, we average over four different splits.

References

Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian D. Reid, Stephen Gould, and Anton van den Hengel. 2018. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 3674–3683. IEEE Computer Society.

Jacob Andreas, Dan Klein, and Sergey Levine. 2017. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175. PMLR.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 39–48. IEEE Computer Society.

Yoav Artzi, Dipanjan Das, and Slav Petrov. 2014. Learning compact lexicons for CCG semantic parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1273–1283, Doha, Qatar. Association for Computational Linguistics.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1726–1734. AAAI Press.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

David L. Chen and Raymond J. Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. 2020. Compositional generalization via neural-symbolic stack machines. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Michele Colledanchise and Petter Ögren. 2018. *Behavior trees in robotics and AI: An introduction*. CRC Press.

Abhishek Das, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. 2019. Neural Modular Control for Embodied Question Answering. *arXiv:1810.11181 [cs]*. ArXiv: 1810.11181.

Matt Deitke, Winson Han, Alvaro Herrasti, Aniruddha Kembhavi, Eric Kolve, Roozbeh Mottaghi, Jordi Salvador, Dustin Schwenk, Eli VanderBilt, Matthew Wallingford, Luca Weihs, Mark Yatskar, and Ali Farhadi. 2020. Robothor: An open simulation-to-real embodied AI platform. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 3161–3171. IEEE.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*.

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

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable programs with neural libraries. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1213–1222. PMLR.

Daniel Gordon, Aniruddha Kembhavi, Mohammad Rastegari, Joseph Redmon, Dieter Fox, and Ali Farhadi. 2018. IQA: visual question answering in interactive environments. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4089–4098. IEEE Computer Society.

Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. 2017. Cognitive mapping and planning for visual navigation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 7272–7281. IEEE Computer Society.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2980–2988. IEEE Computer Society.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vi-

sion and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

Hengyuan Hu, Denis Yarats, Qucheng Gong, Yuandong Tian, and Mike Lewis. 2019. Hierarchical decision making by generating and following natural language instructions. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10025–10034.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5426–5435, Hong Kong, China. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Yacine Jernite, Kavya Srinet, Jonathan Gray, and Arthur Szlam. 2019. CraftAssist Instruction Parsing: Semantic Parsing for a Minecraft Assistant. *arXiv:1905.01978 [cs]*. ArXiv: 1905.01978.

Yiding Jiang, Shixiang Gu, Kevin Murphy, and Chelsea Finn. 2019a. Language as an abstraction for hierarchical deep reinforcement learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9414–9426.

Yu-qian Jiang, Shi-qi Zhang, Piyush Khandelwal, and Peter Stone. 2019b. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, 20(3):363–373.

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

Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. 2020. Learning adaptive language interfaces through decomposition. In *Proceedings of the First Workshop on Interactive and Executable Semantic Parsing*, pages 23–33, Online. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Lars Kunze, Moritz Tenorth, and Michael Beetz. 2010. Putting people's common sense into knowledge bases of household robots. In *Annual Conference on Artificial Intelligence*, pages 151–159. Springer.

Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8198–8210, Online. Association for Computational Linguistics.

Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. 2019. Synthesizing environment-aware activities via activity sketches. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 6291–6299. Computer Vision Foundation / IEEE.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

Chih-Yao Ma, Jiasen Lu, Zuxuan Wu, Ghassan Al-Regib, Zsolt Kira, Richard Socher, and Caiming Xiong. 2019. Self-monitoring navigation agent via auxiliary progress estimation. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. 2018. Mapping instructions to actions in 3D environments with visual goal prediction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2667–2678, Brussels, Belgium. Association for Computational Linguistics.

Dipendra K Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. 2016. Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research*, 35(1-3):281–300.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018. Virtualhome: Simulating household activities via programs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8494–8502. IEEE Computer Society.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Mukund Raghothaman, Y. Wei, and Y. Hamadi. 2016. Swim: Synthesizing what i mean - code search and idiomatic snippet synthesis. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 357–367.

Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10824–10834.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020a. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10737–10746. IEEE.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020b. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. *arXiv:2010.03768 [cs]*. ArXiv: 2010.03768.

Kunal Pratap Singh, Suvaansh Bhambri, Byeonghwi Kim, Roozbeh Mottaghi, and Jonghyun Choi. 2020. Moca: A modular object-centric approach for interactive instruction following. *arXiv preprint arXiv:2012.03208*.

Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 639–646. IEEE.

Siddharth Srivastava, Lorenzo Riano, Stuart Russell, and Pieter Abbeel. 2013. Using classical planners for tasks with continuous operators in robotics. In *Intl. Conf. on Automated Planning and Scheduling*, volume 3. Citeseer.

Alane Suhr, Claudia Yan, Jack Schluger, Stanley Yu, Hadi Khader, Marwa Mouallem, Iris Zhang, and Yoav Artzi. 2019. Executing instructions in situated collaborative interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2119–2130, Hong Kong, China. Association for Computational Linguistics.

Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. 2020. Program guided agent. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211.

Moritz Tenorth, Daniel Nyga, and Michael Beetz. 2010. Understanding and executing instructions for everyday manipulation tasks from the World Wide Web. In *2010 IEEE International Conference on Robotics and Automation*, pages 1486–1491, Anchorage, AK. IEEE.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2692–2700.

Jiajun Wu, Erika Lu, Pushmeet Kohli, Bill Freeman, and Josh Tenenbaum. 2017. Learning to see physics via visual de-animation. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 153–164.

Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica Lam. 2021. Grounding open-domain instructions to automate web support tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1022–1032, Online. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Licheng Yu, Xinlei Chen, Georgia Gkioxari, Mohit Bansal, Tamara L. Berg, and Dhruv Batra. 2019. Multi-target embodied question answering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 6309–6318. Computer Vision Foundation / IEEE.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Luke Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*.

Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. 2019. Rtfm: Generalising to novel environment dynamics via reading. *arXiv preprint arXiv:1910.08210*.

Victor Zhong, Caiming Xiong, and R. Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *ArXiv*, abs/1709.00103.

Fengda Zhu, Yi Zhu, Xiaojun Chang, and Xiaodan Liang. 2020. Vision-language navigation with self-supervised auxiliary reasoning tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10009–10019. IEEE.

A Full Procedural Library

The full procedural library for IQA is listed in Tab. 9 and that for ALFRED is listed in Tab. 10.

B User-issued Procedural Library

Fig. 3 shows the screenshot of the annotation guideline. We purposefully avoid any dataset-related examples. The programmer takes around 90 minutes to complete the annotation. The procedural library created by a programmer without prior knowledge to the ALFRED dataset is in Tab. 11. The programmer could issue reasonable procedural functions that highly resemble our own creations. The reactors can be added to detect the properties of the objects before the condition clauses.

C Experiment Settings

In this section of the appendix, we describe the detailed implementation of the pre-search map, the heuristic induction of supervisions from existing annotation of the AFLFRED dataset and the implementation of the baseline and our HMN for reproduce purpose.

C.1 Pre-search Map Procedure

We treat each scene as a grid map with grid size 0.25. The agent stands on each point, turn around 90 degrees a time and move its camera with degree [-30, 0, 30] and scan. The best position for a receptacle satisfy (1) the agent can open/close the receptacle, can pick up/put an object from/to it. (2) the visual area of the receptacle is the largest compared to other positions. A threshold is used to avoid standing too closed. For ALFRED only, we record the positions of movable objects (*e.g.* apple). This is done by enumerating all the receptacle positions, open them if needed and select the receptacle position that makes the object most visible.

The map creation also requires an object detection model to detect objects for each scan. For IQA, we use the fine-tuned YOLO-v3 detector as describe in §6.1 and the area of an object is calculated by its bounding box. For ALFRED, we instead use an oracle object detector to minimize the pre-search performance loss.

Notably, there are many existing works that apply the similar replacement (Shridhar et al., 2020b; Karamcheti et al., 2020). For example, Shridhar et al. (2020b) pre-search the map, records the co-ordinates of each object and uses an A* planner

to navigate between two positions. This replacement that allows the agent to proceed to a location without fail.

C.2 Reactive Baseline

IQA The reactive baseline is implemented as a pointer network (Vinyals et al., 2015) whose output sequence corresponds to the positions in an input sequence. To make a fair comparison with our method, we provide this baseline with the available receptacle IDs of each scene, the question type, and the targeted objects. For instance, given the question *how many mugs in the fridge* for scene i, we list all the receptacles (*e.g.* fridge_1, cabinet_2) in the order of distance to the agent's initial position as well as the question type "contains" and the two working objects "mug" and "fridge". The fixed set of actions and the answers are added at the beginning of the input so that the model does not need an extra generation component. The reactive agent needs to navigate to each receptacle, operate them properly and generate an answer at the end. The images are encoded and the objects are detected with the same YOLO-v3 detector as in HMN.

While an action sequence is not provided in the release of the dataset, we heuristically create such action sequences by enumerating the input receptacle list of each sample. The size for each question type is 7000 and a total of 21000 samples are used in the training. We additionally compare with the HIMN proposed in Gordon et al. (2018) that designs a meta-controller that calls different controllers to accomplish different tasks (*e.g.* navigation, manipulation), and an A3C agent implemented in the same work.

ALFRED We follow Shridhar et al. (2020a) to setup our reactive baseline. This baseline takes the natural language instruction x as input, then it predicts an atomic action at each time step, conditioned on the vision, the previous generated atomic action, and the attended language. The baseline also has a progress monitor component to track task completion progress (Ma et al., 2019). We make the same replacement of the atomic navigation actions with `Navigate` destination. The original mask generator is replaced by the same Mask R-CNN used in our HMN.

For both datasets, we use `seen` and `unseen` validation set for the evaluation. The floorplans of the `unseen` split are held-out in the training data. Each floorplan defines the appearance of the environment as well as the arrangement of the objects.

Annotation Guideline

Assuming you are creating a library written in Python that could be used to *describe how to accomplish a set of tasks*.

To understand the tasks, you are given 7 task categories and in each category, you are given 3 trajectories to achieve a specific goal stated as natural language. Each trajectory consists of a sequence of *atomic actions(e.g. GotoLocation)* and their *arguments(e.g. Desktop)*.

One key feature of the function you create is **reusable**. For example, if an action sequence (e.g. atomic_action_1, atomic_action_2 and atomic_action_3) is frequently observed, you can compose super_action_1 that consists of these three actions. In addition, you can use any composed super_action to compose other super_actions. For example, if there is a super_action_2 that consists of atomic_action_1, atomic_action_2 and atomic_action_3 and atomic_action_4, you can define this super_action_2 as super_action_1, atomic_action_4. Their corresponding Python functions are listed below. You can freely name the arguments, which can be as simple as 'object_1', 'object_2'

```python
def super_action_1(arg1, arg2):
    atomic_action_1(arg1)
    atomic_action_2(arg2)
    atomic_action_3(arg2)

def super_action_2(arg1, arg2):
    super_action_1(args1, args2)
    atomic_action_4(arg2)
```

Another key feature of the function you create is **good coverage/generalizable**. As in your daily life, you can take different actions to accomplish the same goal. The different action might be due to the diverse nature of accomplishing the task (e.g. you can either order online or go to a local supermarket to buy some food). Or it is due to the dynamic environment (e.g. when you buy the food in the supermarket that only accepts cash, you have to withdraw money if you don't have any, but you can skip this withdrawal process if you have cash with you). This is defined through conditions

```python
def shop_in_super_market:
    if not_have_cash:
        withdraw_cash()
    # shopping, a super_action
    super_action_i()
```

The reason why we treat this function as a more generalizable function is that, if you do not write in this way, you will have to compose two distinct functions even though they achieve the same goal in the end:

```python
def shop_in_super_market_with_cash:
    # shopping

def shop_in_super_market_without_cash:
    withdraw_cash()
    # shopping
```

Figure 3: The annotation guideline for a programmer to create procedural functions with 21 examples from the ALFRED dataset.

For IQA, we measure the answer accuracy, and we follow Shridhar et al. (2020a) to measure the task success rate (SR), which defines the percentage of whole task completion; and sub-task success rate (SSR), which measures the ratio of individual sub-task completion for ALFRED.

C.3 HMN Implementation

IQA Since the natural language questions x are generated with a limited number of templates, we use a rule-based HMN-PLANNER that recognizes each template and classifies a template to one of the three question types whose corresponding procedural actions are listed as the top three functions in Tab. 9.

We model the two reactors ATTRCHECKER and RELCHECKER as two multi-classes classifiers. We first follow Gordon et al. (2018) to use a YOLO-v3 (Redmon and Farhadi, 2018) that is fine-tuned on the images sampled from THOR for object detection. This object detector scan each visual input and generate a bounding box and a class name for each detected object. Since there are only seven receptacles, the ATTRCHECKER uses the predicted class name of a receptacle to decide whether the receptacle is openable or not. It then marks the receptacle as is_open=True after the atomic open action is launched for the receptacle. The RELCHECKER use bounding box to heuristically decide the spatial relation between an object and a receptacle. The RELCHECKER considers that an object is inside a receptacle if its bounding box has over 70% overlap with the receptacle's bounding box.

ALFRED We use a sequence-to-sequence model with attention (Bahdanau et al., 2015) as our PLANNER. The input to the encoder is the natural language x. The decoder generates one function a_i at a time from a constrained vocabulary

$\mathcal{A}^p \bigcup \mathcal{A}^a$, conditioned on x and the action history $\{a_1, ..., a_{i-1}\}$.

We adopt the pre-trained Mask R-CNN (He et al., 2017) that is fine-tuned on the ALFRED dataset from Shridhar et al. (2020b) as our MGENERATOR. It returns the name and the bounding box for all detected objects in the visual input. Its parameters are frozen. We design ATTRCHECKER and REFINDER as two multi-classes classifiers. The inputs to these two reactors are the object name h^o encoded by a BI-LSTM, the immediate vision h^i encoded by a frozen RESNET-18 CNN (He et al., 2016) following Shridhar et al. (2020a), the called action sequence h^a encoded with a LSTM and the attended input h^l with h^a. These four vectors are concatenated together as h^f. A fully connected layer and a non-linear activation function are added to predict class probabilities.

C.4 AFLRED Supervision Induction

We induced the ground truth labels for each component of the HMN from ALFRED with the help of atomic action sequences and the subgoal sequences provided by the dataset so that the HMN can be trained in a supervised fashion to maximize the log-likelihood of the label. First, we used the subgoal sequences to annotate the executable procedural actions for the planner. For example, a subgoal sequence Goto, Pick, Clean, Goto, Put was annotated with udp_clean_object, udp_put_object. A different subgoal sequence Goto, Pickup, Goto Clean, Put was annotated with the same procedural action sequence. The first author annotated 30 most frequent subgoal sequences of the training set of ALFRED and resulted in 19 different executable procedural actions[6]. Next, we used the atomic action sequences of the dataset to generate the labels for the reactors. For example, if there is an Open before a Pickup in the atomic action sequence, the attribute of the corresponding object is labeled as openable=True and is_open=False.

When doing the sanity check to verify the coverage of our created procedural library, we assign an executable procedural action a^e to each sample, we then check whether the atomic action sequence of a^e match the annotated atomic action sequence provided by the dataset. Unmatched examples are reviewed and the procedural library is updated as

---

[6]We discarded a training example if its subgoal sequence is not annotated with the procedure library. About 500 samples among 21k training data are discarded.

in §6.2.

C.5 Hyperparameters

IQA Baseline The embedding size is 100, the hidden size of the BI-LSTM and LSTM are 256 and 512. We take the same three feature vectors before the YOLO detection layer and convert the channel size to 32 with convolution layers to encode an image. The flatted features are concatenated with dropout rate of 0.5. We use Adam (Kingma and Ba, 2015) with learning rate 1e-4.

ALFRED We follow Shridhar et al. (2020a) for the hyperparameter selection of the baseline and our model if they are applicable (*e.g.* embedding size, optimizer). We observe that training longer yields better task completion, and thus we train the baseline for 15 epochs and ours for 10 epochs. For our method only, the size of h^o, h^a and h^l is 512. The activation function of ATTRCHECKER is Sigmoid and the output size is 3 (*i.e.* is_openable, is_open, is_close). The activation function of REFINDER is Softmax and the output size equals the object vocabulary size.

D Analysis

In this section, we present concrete examples to demonstrate the benefit of our proposed pipeline. We also show a few failures of our pipeline to encourage future developments.

D.1 Advantage of HMN

The above results suggest that our proposed framework with modularized task-specific components and predefined procedure knowledge is effective in controlling situated agents via complex natural language commands. Compared with the reactive agent, this framework brings several benefits. First, instead of directly controlling an agent using low-level atomic actions, it predicts holistic procedural programs, which are better aligned with high-level input NL descriptions. For instance, in Examples 1 and 2 in Tab. 8, common NL phrases like *put · in ·* naturally map to the procedure udp_pick_put_object, while the reactive baseline could struggle at interpreting the correspondence between the NL intents and the verbose low-level atomic actions, resulting in incomplete predictions. Second, using procedures could help maintain *consistency* of actions. Specifically, given a procedure (*e.g.* udp_pick_put_object), and its

Task: Put a chilled egg in the sink	
Reactive: Navigate egg Pickup egg Navigate fridge Open fridge STOP	
HMN-Planner: udp_cool_object(egg), udp_pick_put_object(egg, sink)	
Task: Put CDs in a safe. (*requires to put *two* CDs)	
Reactive: Navigate cd Pickup cd Navigate safe Open safe Put cd safe Close safe STOP	
HMN-Planner: udp_pick_put_object(cd, safe), udp_pick_put_object(cd, safe)	
Task: Place a cooked potato slice in the fridge	
Reactive: Navigate knife Pickup knife Navigate potato Slice potato Navigate fridge Put knife countertop Navigate potato Close potato ...	
HMN-Planner: udp_slice_object(potato), udp_heat_object(potato), udp_pick_and_put(potato, fridge)	

Table 8: Common failures of the reactive baseline. All actions of the reactive baseline are atomic actions.

arguments (*e.g.* knife, fridge), the HMN agent is guaranteed to coherently carry out the specified action without being interfered, while the reactive baseline could predict inconsistent atomic actions in-between (*e.g.* the underscored arguments of Navigate and Put should be the same in Example 3). Finally, we remark that procedures also improve the robust behavior of the agent. For instance, when interacting with container objects (*e.g.* fridge), HMN would call the dedicated AT-TRCHECKER to decide whether to open the object first (*e.g.* C4, Fig. 1), and it mis-predicts once, while the reactive baseline fails to perform the Open action 33 times on the *unseen* split.

D.2 Error Analysis

We first did an ablation study on the PLANNER on the *unseen* split. PLANNER correctly predicts 80% executable procedural actions a^e, and the failures are mainly due to rare words in utterances (*e.g.* soak a plate. Next, we manually annotated 50 failed examples among samples whose a^e are correctly predicted by the PLANNER. We found that 26 failures are due to the sub-optimal interaction positions of the receptacles that we compute during the pre-search phase (§C.1). This results in the failures of putting an object in-hand to a visible receptacle or picking up a visible object. The pre-search map also missed some objects and navigating to these objects always failed. This problem can be alleviated either by adding additional procedural actions to move around and attempt to pick up or put an object until success, or by doing more careful engineering to create the map. Additionally, 18 examples are caused by prediction errors of reactors. For instance, REFINDER could given incorrect predictions of the containing receptacle of an object. The receptacle is not correctly operated before the targeted object is visible. While such errors are inevitable due to imperfect reactors, it could be potentially mitigated by designing more robust procedures, *e.g.*, enumerating over the

top-n most likely receptacles for a target object instead of the best scored one by the reactor. Other approaches, like introducing object-centric representations to the reactors (Wu et al., 2017; Singh et al., 2020), could also be helpful. The remainder of the errors are caused by ambiguous annotation (2 examples), and wrong argument predictions of the planner (4 examples).

E Related Work

Procedure-guided Learning The idea of using predefined procedures for agent control has been explored in the literature. For example, Andreas et al. (2017); Das et al. (2019) use high-level symbolic program sketches to guide an agent's exploration; Gordon et al. (2018); Yu et al. (2019) design meta-controller to call different low-level controllers. There only exists one explicit level of the hierarchy. Sun et al. (2020) show that programs can assist agent's task completions. They require the presence of the program for each task, while our programs are generated by the planner. There is no nested function in their provided programs too. Programs are used to represent procedures in Puig et al. (2018), but no hierarchy is considered. Later Liao et al. (2019) annotate the dataset with program sketches and propose a graph-based method to generate executable programs. Their work requires a fully observed environment while we only consider egocentric visions. Recent works also explore representing hierarchies with natural language (Hu et al., 2019; Jiang et al., 2019a) and visual goal representation (Misra et al., 2018) instead of symbols. Another related area is probabilistic programming, where procedures serve as symbolic scaffolds to define the control flow of learnable programs (Gaunt et al., 2017). Our work is related to these research in using predefined procedural knowledge to assist learning, while we focus on leveraging such procedures to synthesize executable programs from natural language commands.

Semantic Parsing Our work is also related to semantic parsing, where executable programs are generated from natural language inputs. This includes mapping NL to domain-specific logical forms (*e.g.* lambda calculus, (Zettlemoyer and Collins, 2005)) or programs (*e.g.* SQL, (Zhong et al., 2017; Yu et al., 2018)). Recently there has also been a burgeoning of developing models that could transduce natural language intents into general-purpose programs (*e.g.* Python, (Yin and Neubig, 2017; Rabinovich et al., 2017)). Our work also considers program generation from NL, with a focus on the command and control of situated agents.

Research in semantic parsing has also explored leveraging idiomatic program structures, which are fragments of programs that frequently appear in training data, to aid generation (Raghothaman et al., 2016). Such idiomatic programs are mined from corpora (Iyer et al., 2019; Shin et al., 2019). Our work focuses on designing flexible and idiomatic procedures which interact with situated components (*e.g.* reactors) to adapt to environment-specific situations. This work also uses manually-curated procedures, because in our problem setting we do not have a readily available corpus of high-level procedural programs to automatically collect such idioms. We leave extracting procedures from low-level atomic actions as interesting future work.

Robotics Planning and Hierarchical Control Our procedure library shares the design philosophy with the macro-actions in the STRIPS representation in the robotics planning (Fikes and Nilsson, 1971). However, we do not define the pre-condition and the post-effect of the actions, and instead leave the models to learn the consequences. The task-level planning has been studied extensively (Kaelbling and Lozano-Pérez, 2011; Srivastava et al., 2013, 2014). These methods often work with high-level formal languages in low-dimensional state space, and they are typically designed for a specific environment and task. Our framework can be applied to various tasks and only partial observations are required. Previous works also leverage PDDL and the answer set planner (ASP) for task planning. PDDL+ASP is conceptually different from our formalism. PDDL+ASP aims at planning the actual execution sequences. The PDDL planner searches the action sequences based on the initial and the final state. Meanwhile, our formalism focuses on describing the procedure to accomplish

a task. We use the HMN-Planner to predict the executable procedure sequence given the NL. It is possible to integrate them into one system. E.g.,a procedure function can call a PDDL planner if the pre/post conditions are clearer given NL. Finally, many works design mechanism to learn hierarchies automatically from supervisions of only the end-task (Sutton et al., 1999; Bacon et al., 2017), which might suffer from collapsing to trivial atomic actions.

```python
# check the existence of an object in the scene
def udp_check_obj_exist(obj):
    all_recep = udp_grid_search_recep()
    for recep in all_recep:
        rel = udp_check_relation(obj, recep)
        if rel == OBJ_IN_RECEP:
            return True
    return False


# check whether a receptacle contains an object
def udp_check_contain(obj, recep):
    all_recep = \
    udp_grid_search_tar_recep(recep.desc)
    for recep in all_recep:
        rel = udp_check_relation(obj, recep)
        if rel == OBJ_IN_RECEP:
            return True
    return False


# count how many objects in the scene
def udp_count_obj(obj):
    tot = 0
    all_recep = udp_grid_search_recep()
    for recep in all_recep:
        rel = udp_check_relation(obj, recep)
        tot += int(rel == OBJ_IN_RECEP)
    return tot


# check object inside receptacle
def udp_check_relation(obj, recep):
    atomic_navigate(recep)
    r1 = get_reactor("check_obj_attr")
    r2 = get_reactor("check_obj_recep_rel")
    attr = r1(recep)
    if attr.is_openable and attr.is_closed:
        atomic_open_object(recep)
        rel = r2(obj, recep)
        atomic_close_object(recep)
    else:
        rel = r2(obj, recep)
    return rel


# get a list of target receptacles
def udp_grid_search_tar_recep(desc):
    recep_list = udp_grid_search_recep()
    tar_recep_list = [x for x in recep_list \\
    if x.desc == desc]
    return tar_recep_list
# navigate and search at every reachable points
def udp_grid_search_recep():
    if not done_search:
        all_receps = [] # global var
        for pos in reachable_pos:
            atomic_navigate_pos(pos)
            all_receps += udp_search_recep()
    return all_receps
```

Table 9: Procedural functions defined for IQA

```
# close an object if it is open
def udp_close_if_needed(obj):
    reactor = get_reactor("check_obj_attr")
    attr = reactor(obj)
    if attr.is_openable and attr.is_open:
        atomic_close_object(obj)
# "postpare" the receptacle
def udp_postpare_recep(obj):
    reactor = get_reactor("check_obj_attr")
    attr = reactor(obj)
    if attr.is_openable and attr.is_open:
        atomic_close_object(obj)
# pickup an object
def udp_pick_object(obj):
    reactor = get_reactor("find_obj_recep")
    udp_navigation(obj)
    recep = reactor(obj)
    udp_prepare_recep(recep)
    # this is for pickup laptop only
    udp_close_if_needed(obj)
    atomic_pick_object(obj)
    udp_postpare_recep(recep
# put an object to a receptacle
def udp_put_object(obj, dst):
    udp_navigation(dst)
    udp_prepare_recep(dst)
    atomic_put_object(obj, dst)
    udp_postpare_recep(dst)
# clean an object in the fauucet
def udp_clean_object(obj):
    # sink and faucet are global variables
    udp_pick_object(obj)
    udp_put_object(obj, sink)
    atomic_toggleon_object(faucet)
    atomic_toggleoff_object(faucet)
    udp_pick_object(obj)
# slice an object with a knife
def udp_slice_object(obj, tool_dst):
    # knife is a global variable
    udp_pick_object(knife)
    udp_navigation(obj)
    reactor = get_reactor("find_obj_recep")
    recep = reactor(obj)
    udp_prepare_recep(recep)
    atomic_slice_object(obj)
    udp_postpare_recep(recep)
    udp_put_object(tool, tool_dst)
# pick an object and then put it to a receptacle
def udp_pick_and_put_object(obj, dst):
    udp_pick_object(obj)
    udp_put_object(obj, dst)
# cool an object with fridge
def udp_cool_object(obj):
    # fridge is a global variable
    udp_pick_and_put_object(obj, fridge)
# heat an object with microwave
def udp_heat_object(obj):
    udp_pick_and_put_object(obj, microwave)
    atomic_toggleon_object(microwave)
# prepare a receptacle for interaction
def udp_prepare_recep(obj):
    reactor = get_reactor("check_obj_attr")
    attr = reactor(obj)
    if attr.is_openable and attr.is_closed:
        atomic_open_object(obj)
```

Table 10: Procedural functions defined for ALFRED

```
# udp_pick_object(obj):
def udp_pick_up(object, loc):
    udp_navigation(loc)
    if loc.is_open:
        atomic_pickup_object(object)
    else:
        atomic_open_object(loc)
        atomic_pickup_object(object)
        atomic_close_object(loc)


def udp_pick_up_to(object, loc, loc_to):
    udp_pick_up(object, loc)
    udp_navigation(loc_to)


# udp_put_object(obj, dst):
def udp_put_to(object, loc_to):
    udp_navigation(loc_to)
    if loc.is_open:
        PutObject(object)
    else:
        atomic_open_object(loc_to)
        atomic_put_object(loc_to)
        atomic_close_object(loc_to)


# udp_pick_and_put_object(obj, dst):
def udp_pick_put_to(object, loc, storage):
    udp_pick_up(object, loc)
    udp_put_to(object, storage)


def udp_look_under_light(object, loc, light_source):
    udp_pick_up_to(object, loc, light_source)
    atomic_toggleon_object(light_source)


# udp_slice_object(obj, tool_dst):
def udp_slice(object, loc, slicer):
    udp_pick_up_to(slicer, loc, object)
    atomic_slice_object(object)


def udp_toggle(object):
    atomic_toggleon_object(object)
    atomic_toggleoff_object(object)


# udp_cool_object(obj):
def udp_cool(object, loc):
    udp_pick_put_to(object, loc, fridge)


# udp_heat_object(obj):
def udp_heat(object, loc):
    udp_pick_put_to(object, loc, microwave)
    udp_toggle(microwave)
    udp_pick_up(object, microwave)


# udp_clean_object(obj):
def udp_clean(object, loc):
    udp_pick_put_to(object, loc, Faucet)
    udp_toggle(Faucet)
```

Table 11: Procedural functions defined by a programmer without ALFRED domain knowledge. The comments could roughly map to functions in Tab. 10.