

# Synthesizing Environment-Aware Activities via Activity Sketches

Yuan-Hong Liao<sup>1,2\*</sup>, Xavier Puig<sup>3\*</sup>, Marko Boben<sup>4,5</sup>, Antonio Torralba<sup>3</sup>, Sanja Fidler<sup>1,2</sup>

<sup>1</sup> University of Toronto, <sup>2</sup> Vector Institute, <sup>3</sup> MIT, <sup>4</sup> University of Ljubljana, <sup>5</sup> Abelium d.o.o  
 {andrew, fidler}@cs.toronto.edu, {xavierpuig, torralba}@csail.mit.edu, marko.boben@fri.uni-lj.si

## Abstract

In order to learn to perform activities from demonstrations or descriptions, agents need to distill what the essence of the given activity is, and how it can be adapted to new environments. In this work, we address the problem of environment-aware program generation. Given a visual demonstration or a description of an activity, we generate program sketches representing the essential instructions and propose a model to transform these into full programs representing the actions needed to perform the activity under the presented environmental constraints. To this end, we build upon VirtualHome [17] to create a new dataset VirtualHome-Env, where we collect program sketches to represent activities and match programs with environments that can afford them. Furthermore, we construct a knowledge base to sample realistic environments and another knowledge base to seek out the programs under the sampled environments. Finally, we propose ResActGraph, a network that generates a program from a given sketch and an environment graph and tracks the changes in the environment induced by the program.

## 1. Introduction

We want agents to be able to perform everyday tasks (such as setting up the table, preparing coffee, or even sit on the couch and watch TV). An agent should learn to perform these tasks from high-level descriptions or visual demonstrations. The challenge is on how to generalize the acquired knowledge to new environments. For instance, if we want to learn to make coffee, an agent could first watch a video of someone making coffee in order to extract the sequence of steps (program) that need to be executed. However, when trying to make coffee in an environment that differs from the environment in which the demonstration took place, the agent has to adjust the program so that it can be executed. For instance, the agent could find that the coffee machine is unplugged, or that it is in the living room instead

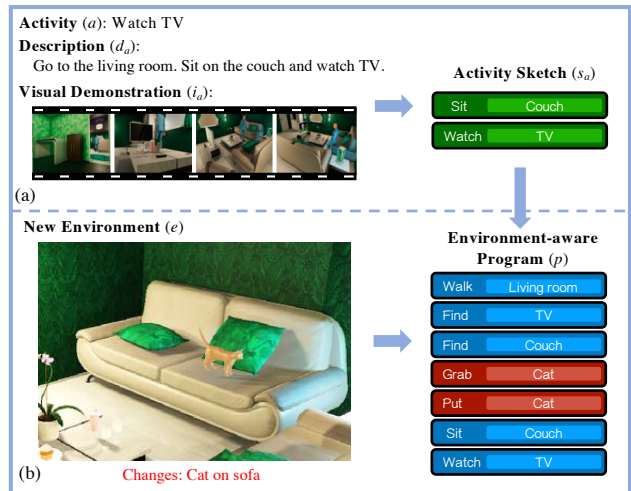


Figure 1. Overview of the *environment-aware program generation*. Our goal is (a) generating a sketch  $s_a$  distilling the essential steps of the given demonstration  $i_a$  or description  $d_a$  and (b) given a new environment  $e$ , generating a program  $p$ , adapting the sketch to  $e$ . The program contains the instructions to perform the activity (blue blocks) as well as instructions to deal with the environment (red blocks, grabbing the cat to sit).

of in the kitchen, or that someone else is currently using it and the agent needs to wait. Being able to perform these adjustments requires the access to a common-sense database of knowledge that allows the agent to decide which steps in the demonstration are essential in the task definition, and how the program needs to be modified (by adding/removing steps) in order to accomplish the task in a new environment.

To address the generalization problem we represent activities with sketches, representations inspired from work in programming languages [18, 15]. Sketches are high-level representations of the steps needed to perform a task but leaving holes that need to be completed for the sketch to become executable in a particular environment.

Fig. 1 illustrates our goal. Given a visual demonstration or a description of someone going to watch TV, we want to extract the sketch of the activity (fig. 1a). In this example, the sketch consists of two steps: Sit Couch, Watch TV. The fact that the demonstration had the person going to the

\* indicates equal contribution

living-room is not important. Watching TV requires us to go to the room that contains the TV. Executing this activity in a new environment requires expanding the sketch into a complete program (fig. 1b). The sketch expansion depends on the state of the environment. In this example, it turns out that there is a cat on the couch. To sit on the couch, we need to push away the cat first. We show the automatically generated program that includes all the steps needed to complete the task and to deal with the cat.

To expand sketches into programs, it is necessary to have access to commonsense knowledge about the world and how to deal with typical situations. This knowledge provides ways to solve situations that can be reused in a multitude of tasks. Since no such information, with diverse programs and environments, exists to date, we collect a new dataset, built upon VirtualHome [17], containing around 3k home activities. We extend the VirtualHome dataset to include more actions and over 30k programs and collect sketches of the activities by crowdsourcing. Each environment is represented as a graph with 300 objects and 4000 spatial relations on average.

We then propose a model to generate programs by selecting, for every step, a node in the environment graph representing an object of interaction. To do so, we exploit Graph Neural Networks (GNNs) to reason about the states and relations between the objects in the environments. Furthermore, we propose ResActGraph, a model that reasons about the changes in the graph induced by the agents previous steps to generate the goal program.

The main contributions of this work are: introducing sketches as environment-independent representations of an activity, a database of commonsense knowledge of activities, sketches and how to deal with a variety of situations, and a method to generate programs from sketches that accomplish the task in a new environment. We show results in VirtualHome [17].

## 2. Related Work

**Learning from demonstrations.** Learning from visual demonstrations or language descriptions has been of increasing interest in both robotics and computer vision. However, this has been mostly focused on learning low-level tasks rather than the high semantic level activities that we tackle in this work. For example, [20, 13, 1] focus on learning to navigate in environments or manipulate objects, while in visual imitation learning, multiple works have used videos to learn to manipulate objects under low supervision regimes [16, 7] or imitate kinetic human behaviors [14]. Our work is more closely related to semantic planning [10, 22, 3], which focus on modeling sequences of composite and semantically loaded actions. Learning those requires inferring and modeling the sub-goals of a given task. [10] represents such goals as a graph, with nodes be-

ing actions and edges being precondition states, while we propose program representations. Furthermore, to perform such tasks, it is necessary to know what the constraints of the given environment where it will be executed are. Similarly to [3], we propose to encode knowledge of the environments and how they can constrain the activities to be performed.

**Program Synthesis by Sketching.** Our approach for environment-aware program generation is partially inspired by performing program synthesis by sketching. In [18], a sketch expresses the high-level structure of an implementation but leaves holes in place of low-level details, which corresponds to our model that derives the details based on the environmental constraints so as to execute the programs smoothly. A recent body of work has developed neural approaches to program generation using user-provided examples [8, 4], visual demonstrations [19], and descriptions [6]. The work [15] is the most related to ours. They learn a model that predicts sketches of programs relevant to a label and the predicted sketches are concretized into code using combinatorial techniques. The main difference between our work and theirs is not only that our sketches are inferred from visual or textual data, but that we focus on how to incorporate the environmental constraints in program generation.

## 3. Problem formulation

The goal of environment-aware program generation is to predict, given a demonstration or description of an activity and an environment, a program that can execute the activity in such environment. We define this task and the corresponding notation in this section.

Let  $A$  and  $E$  be the universe of activities and environments. An activity  $a$  (e.g. watch TV) can be represented as a set of programs  $P_a$  containing a sequence of instructions (e.g. TurnOn TV, Sit Sofa, Watch TV), which vary depending on how the activity is performed. Let  $\mathbb{1}(p, e)$  be an indicator function determining if  $p$  can be executed in  $e$  (e.g. an agent can not grab cups inside a closed cabinet). Given an activity  $a \in A$  and an environment  $e \in E$ , our goal is to learn a model that generates  $p$  such that

$$p \in P_a, \mathbb{1}(p, e) = 1 \tag{1}$$

We use the corresponding visual demonstrations  $i_a \in I_a$  or descriptions  $d_a \in D_a$  to specify  $a$ . However,  $i_a$  and  $d_a$  are implicitly conditioned on certain environments which might differ from the current one  $e$ . Therefore, directly inferring  $\hat{p}$  does not ensure that  $\hat{p}$  satisfies eq. 1 given  $e$ .

Inspired by [18], we introduce program sketches  $s_a \in S$  as *environment-independent* representations of the activities. We thus change the constraint in eq. 1 to be:

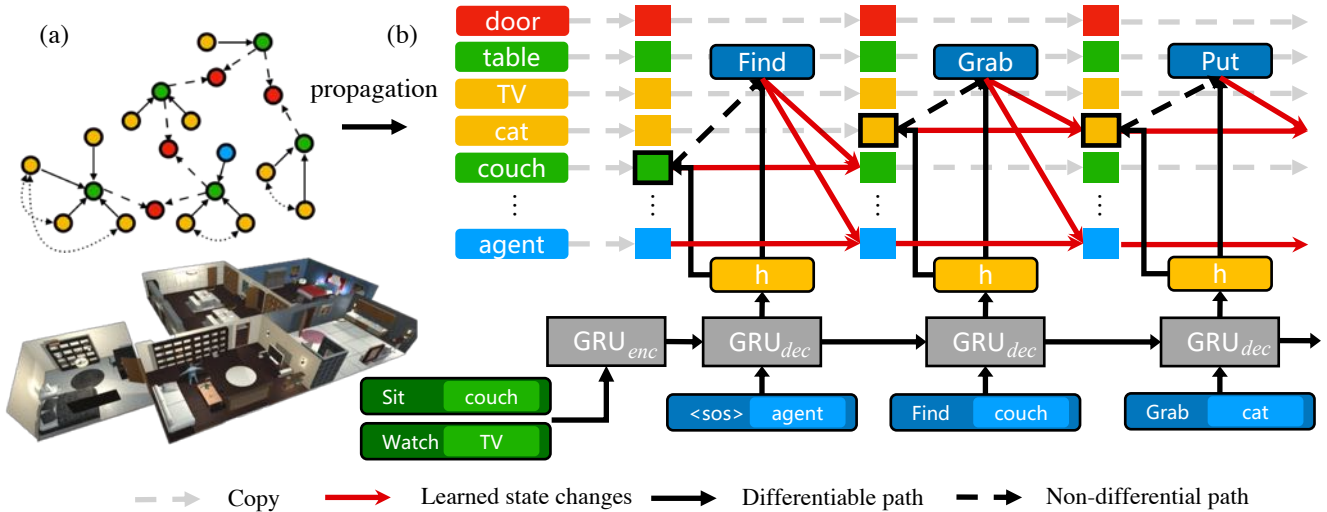


Figure 2. (a) We extract the ground truth environment graph from VirtualHome and perform message passing on the graph. (b) At every time step, the decoder perform sequential classification over the hidden states of the graphs (top row). The selected nodes are shown in bold border blocks. We also model the environment changes induced by the generated programs (solid red arrows).

$$p \in P_{s_a}, \mathbb{1}(p, e) = 1 \quad (2)$$

With the program sketches as proxy representations, we can divide the task into two sub-problems: a model that predicts a sketch  $\hat{s}_a$  from a demonstration  $i_a$  or a description  $d_a$  and another model that predicts a program  $\hat{p}$  given the predicted sketch  $\hat{s}_a$  and an environment  $e$ :

$$\begin{aligned} \hat{p} &= f_{sketch2prog}(\hat{s}_a, e), \text{ where} \\ \hat{s}_a &= f_{demo2sketch}(i_a) \text{ or } \hat{s}_a = f_{desc2sketch}(d_a) \end{aligned} \quad (3)$$

Here,  $p$  and  $s_a$  are a sequence of instructions. Each instruction is represented by an action and up to two arguments representing objects of interaction  $(\alpha, \beta^1, \beta^2)$ .<sup>1</sup>

## 4. Model

In this section, we present our approach to the *environment-aware program generation* task. First, we introduce *ResActGraph* to generate programs from sketches and target environments. Later on, we describe how we predict sketches from demonstrations or descriptions.

### 4.1. Program generation from sketches and graphs

We frame the program generation task as a seq2seq problem, where an encoder encodes the input sketch and the decoder generates the target program one instruction at a time, composed by an action and object arguments. Given that

<sup>1</sup>The number of arguments depends on the type of the action.

the program must be grounded in a target environment, instead of predicting the objects from a fixed taxonomy, the model predicts for each instruction object instances that are present in the environment. This has two benefits: (1) It avoids referring to object instances that do not exist in the environment. (2) It allows the model to use information of each instance within the environment, such as its state or relations with other objects, to predict the appropriate instruction.

To do that, we encode the scene as a graph  $G = (\mathcal{V}, \mathcal{R})$  modeling the dependencies of the object instances. The node  $v \in \mathcal{V}$  indicates the object instance and each node has a label, including the object class  $c_v$ , its states  $l_v$ , and properties  $prop_v$ . Note that  $\mathcal{V}$  includes a node for the agent itself. The edge  $r \in \mathcal{R}$  encodes the spatial relations, including ON, IN\_OBJ, IN\_ROOM, CLOSE\_TO, and FACE\_AT, between every two object instances.

The node labels and relations are used to obtain vector embeddings for each instance which are used by the decoder to predict the environment-aware program, as we describe in the following section.

### 4.2. ResActGraph

We adopt the GGNN [12] framework to obtain the hidden states of the nodes and capture the object relations in the environment graph. The hidden states of each node  $v$  are initialized by its label  $(c_v, l_v, prop_v)$ :

$$h_v^0 = \tanh(g_{init}([W_c c_v, W_l l_v, W_{prop} prop_v])) \quad (4)$$

We apply one-hot encoding to the label and set

$W_c, W_l, W_{prop}$  as learnable weights.  $g_{init}$  is a network composed of fully connected layers that combine all the information.

At propagation step  $k$ , each node’s incoming information  $x_v^k$  is determined by aggregating the hidden states of its neighbors  $v' \in \mathcal{N}(v)$  at the previous step  $k-1$ :

$$x_v^k = \sum_{j \in L(\mathcal{R})} \sum_{v' \in \mathcal{N}_j(v)} W_{p_j} h_{v'}^{k-1} + b_{p_j} \quad (5)$$

$L(\mathcal{R})$  denotes the set of edge labels and the linear layer  $W_{p_j}$  and bias  $b_{p_j}$  are shared across all nodes.

After aggregating the information, the hidden states of the nodes are updated through a gating mechanism similar to Gated Recurrent Unit (GRU) [5] as follows:

$$\begin{aligned} z_v^k &= \rho(W_z x_v^k + U_z h_v^{k-1} + b_z), \\ r_v^k &= \rho(W_r x_v^k + U_r h_v^{k-1} + b_r), \\ \hat{h}_v^k &= \tanh(W_h x_v^k + U_h (r_v^k \odot h_v^{k-1}) + b_h), \\ h_v^k &= (1 - z_v^k) \odot h_v^{k-1} + z_v^k \odot \hat{h}_v^k \end{aligned} \quad (6)$$

This results in a vector embedding for each object  $h_v^k$ , with information about its state and relationship with the environment.

We use one GRU to encode the sketches and another one to generate the program one instruction at a time. For time  $t$ , let  $feat_{s_a} = enc(s_a)$  be the output of the sketch encoder, and  $h_{dec}^t$  the hidden states of the decoder. To predict the program instruction,  $(\hat{\alpha}_t, \hat{\beta}_t^1, \hat{\beta}_t^2)$ , we predict the first object argument  $\beta_t^1$  over the graph nodes, use it to predict the action  $\hat{\alpha}_t$  and combine this information to predict the second argument  $\hat{\beta}_t^2$ :

$$\begin{aligned} \hat{\beta}_t^1 &= \arg \max_{v \in \mathcal{V}} \sigma(g_{\beta^1}(h_{dec}^t, h_v^K, feat_{s_a})) \\ \hat{\alpha}_t &= \arg \max_{\alpha \in \mathcal{A}} \sigma(g_{\alpha}(h_{dec}^t, h_{\hat{\beta}_t^1}^K, feat_{s_a})) \\ \hat{\beta}_t^2 &= \arg \max_{v \in \mathcal{V}} \sigma(g_{\beta^2}(h_{dec}^t, h_v^K, feat_{s_a}, h_{\hat{\beta}_t^1}^K, \hat{\alpha}_t)) \end{aligned} \quad (7)$$

where  $\mathcal{A}$  is all the possible actions and  $\sigma$  denotes the softmax function.

Note that so far the hidden states of the nodes  $h_v^K$  are constant over  $t$ , but we would like them to change according to the program being executed. To do that, at time  $t$ , we use the previously generated instructions  $(\alpha_{<t}, \beta_{<t}^1, \beta_{<t}^2)$  to update the hidden states of the nodes. We set the initial state of each node as  $h_v^{K_0} = h_v^K$  and update the state  $h_v^{K_t}$  at time  $t$  if  $v$  is interacted by the agent at the previous time step or is the agent itself. For example, if the instruction at the previous time step is `grab mug`, we use the action embeddings of `grab` to change the hidden states of agent

and `mug`. Let  $\hat{v}$  correspond to the agent node or one of the previous arguments  $\beta_{t-1}^1, \beta_{t-1}^2$ . The state  $h_{\hat{v}}^{K_{t-1}}$  is updated as follows:

$$h_{\hat{v}}^{K_t} = h_{\hat{v}}^{K_{t-1}} + r \quad (8)$$

$$r = \tanh(g_{res}(h_{\hat{v}}^{K_{t-1}} \odot g_e(Emb(\alpha_{t-1}), m_{t-1})))$$

where  $Emb(\alpha_{t-1})$  is the embedding of  $\alpha_{t-1}$  and  $m_{t-1}$  is a one-hot encoding denoting if  $v$  is the subject or the object of  $\alpha_{t-1}$ .  $g_e$  and  $g_{res}$  consider the change of the  $h_{\hat{v}}^{K_{t-1}}$  and predict the residuals. Note that since the node `agent` is involved at every time step, it tracks the progress through the generation. The model overview is shown in Fig. 2.

**Learning.** We use the cross-entropy loss function for program prediction. The GGNN and GRUs are then trained with the back-propagation through time (BPTT).

### 4.3. Inferring sketches

**Activities specified by demonstrations.** We use key frames  $i = [i_n]_{n=1:N_{demo}}$  as the representations of the demonstrations, where  $N_{demo}$  is the length of the key frames. To be specific, we take the bird-eye view of each  $i_n$ . Besides, we also use the ground truth semantic segmentation map  $i_{seg} = [i_{seg_n}]_{n=1:N_{demo}}$  as input. Two CNNs are used to extract the features separately, and we apply late fusion to extract the  $n^{th}$  visual features  $feat_n$  as follows:

$$feat_n = g_{fuse}([CNN_i(i_n), CNN_{seg}(i_{seg_n})]) \quad (9)$$

where  $[, ]$  denotes concatenation. Later on, we max-pool the features over the different steps time steps and apply a GRU to decode the sketches.

**Activities specified by descriptions.** We adopt the seq2seq model with a GRU encoding each word in the description and a GRU to decode each of the sketch instructions.

## 5. Dataset: VirtualHome-Env

Our goal is to generate a sketch  $s_a$  from a demonstration  $i_a$  or description  $d_a$  and induce a program  $p$  from  $s_a$  and a target environment  $e$ . In this section, we describe how we obtain the dataset containing  $(a, i_a, d_a, s_a, e, p)$ . We briefly introduce VirtualHome as the playground for the dataset collection, we describe how we extend the dataset with activity sketches and finally describe how we pair the programs with arbitrary environments where they can be performed, motivating the creation of a common sense knowledge base of activities<sup>2</sup>.

**Background.** VirtualHome is a dataset and simulator to represent human household activities. The dataset was collected by asking annotators to come up with various activities and provide a description for them. In a second

<sup>2</sup>The details of the collected knowledge base are described in 8.6



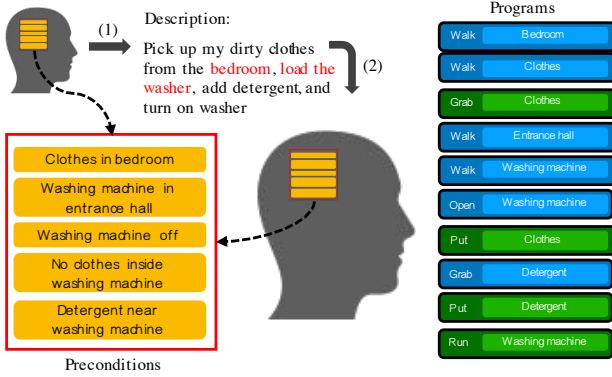


Figure 3. The annotators label the descriptions and programs with certain environments in mind (bottom left), resulting in the *environment-dependent* descriptions and programs. The blocks colored in blue are considered as environment-dependent components.

stage, annotators were shown a description of an activity and were asked to write a program from it. With activities represented as programs, a Unity simulator executes them in some predefined apartments and renders the programs as videos. This allows us to obtain activities and programs together with descriptions and demonstrations  $(a, i_a, d_a, p)$ .

### 5.1. Collecting sketches

When collecting each activity in VirtualHome, annotators imagine an environment where the activity could take place and provide a description according to it. As a result, the description and subsequent program is specific to a given environment, but may not be doable when presented with new environments or constraints (see Fig. 3). Therefore, we need a more abstract representation of an activity, which can be consistent with multiple environments and the information  $(a, i_a, d_a, p)$ .

Inspired by [18], we collect the sketches of the activities to abstract out the components that are environment-dependent and informally define the sketches as the *environment independent representations*. Different from programming languages, it is highly non-trivial to define the sketches of the activities since they depend on the common-sense of each individual. Therefore, we manually collect the sketches and get the information  $(a, i_a, d_a, s_a, p)$ .

### 5.2. Pairing programs with environments

We finally need to add an environment that pairs with the activity programs. Since we do not know the environment that each annotator had in mind, we need to infer it from the program. We first extract the preconditions of the programs and use them to sample feasible environments. We define preconditions of a program as the conditions that have to be true in the environment in order to execute the program in it.

For example, in order to execute `watch TV`, the tv should be on. We construct a function  $\Phi$  that infers the preconditions from  $p$  and sample  $e$  from the set of environments that satisfy such preconditions  $E_{\Phi(p)}$ :

$$e \sim E_{\Phi(p)} \subset E, \text{ s.t. } \forall j \in \Phi(p), e \text{ satisfies } j \quad (10)$$

The above is a weak constraint since it does not inform about objects that are not specified in  $\Phi(p)$ . To get realistic environments, these objects should follow some priors. For example, couches can be occupied, but they can not be cold. Apples can be stored inside fridges, but they are seldom found in bathtubs. We build collect these rules in a knowledge base, *KB-RealEnv* and use them to build the environment, starting from the environments provided in VirtualHome.

### 5.3. Extending programs to diverse environments

We now have the information  $(a, i_a, d_a, s_a, e, p)$ . However, relying solely on the original programs results in a limited set of preconditions and thus environments. One possible reason is that when describing activities, annotators tend to assume the simplest setting to perform the given activity. For example, when thinking of doing the laundry, it is common to imagine that the washing machine is idle or empty. To address that, we build a simulator  $\Psi$  that takes a program  $p$  and environment graph  $e$  and outputs the graph corresponding to the environment after executing the program, or raises an exception if the program is not executable at a certain step. Given a program with preconditions  $\Phi(p)$ , we start by randomly perturbing them into  $\Phi(p)'$  and use eq. 10 to obtain  $e'$  as an environment satisfying  $\Phi(p)'$ . Then, we execute  $p$  in the simulator with the environment  $e'$ . Given that the environment and preconditions have changed but the program is still the same, as the program is executed, some exception will be raised from the simulator. Then, a subroutine is called to modify the program  $p$  into  $p'$ , by inserting or removing instructions to correct the exception, obtaining the extended program.

For example, when we executing `Sit Sofa`, if `Sofa` is occupied, the subroutine is expected to perform actions to remove things on the `Sofa` until there is enough space to `Sit`. We manually compose the subroutines based on different types of exception, forming the knowledge base, *KB-ExceptionHandler*. We show more details of how we augment the programs in the supplementary materials.

This way, we augment over 30k tuples of sketches, environments, and programs  $(s_a, e', p')$ . Note that the sketches  $s_a$  are environment-independent, so there is no need to change them after applying the subroutines.

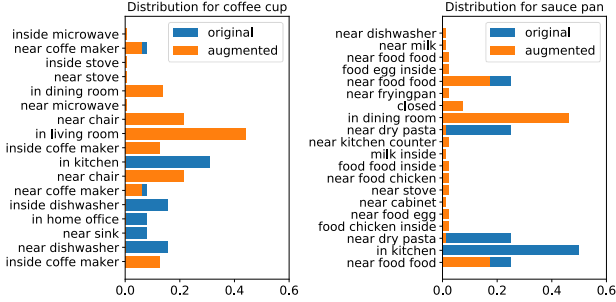


Figure 4. The effect of the dataset augmentation: changes in the distribution of preconditions for the objects in the environment.

## 5.4. Dataset Analysis

From the original 2807 programs in VirtualHome, we trim out the programs that can not be executed in the simulator with the environment sampled via preconditions, obtaining 1387 executable programs. Using the process described in Sec. 5.3, we extend these programs to our final dataset with around 30k programs. The significant increase in the program length is induced by the modified preconditions. For example, the agent needs to open containers to reach objects or make space to sit on a sofa. In Table 6, we show the statistics of the new dataset. Fig. 4 shows the change in the distribution of preconditions of the cup and the sauce pan. The programs after augmentation show a less skewed distribution of preconditions and therefore more diverse environments. Finally, we use the simulator to generate snapshots of the environment after executing each instruction of a program, as show in fig. 12. Note that some of the objects in the program do not have a model in the simulator, so we generate frames for a subset of 8421 programs.

## 6. Experiments

We split the dataset into train and test set in terms of different types of activities with ratio 7:3 and leave one apartment for the test set. We aim to test the capability of our model with novel *activities* and *environments*. We follow the same split for sketch prediction, where we only keep the original programs for the *desc2sketch*, since they contain the collected descriptions, and use the available frames for the *demo2sketch* task.

In this section, we describe the evaluation metrics, baselines. Next, we show the extensive experiment results of *ResActGraph*. Finally, we analyze the extent to which the proposed method is *environment-aware*. We will describe the implementation details in supplementary materials.

### 6.1. Evaluation Metrics

We analyze the performance of sketch prediction and program generation by measuring the normalized longest common subsequence (LCS) between the generated and

ground truth sequences. LCS is sensitive to the order of the sequences and allows gaps in between. To further measure if the generated programs achieve the specified activities, we compute the differences between the final environment graphs  $\hat{G} = \Psi(\hat{p}, e)$  and  $G = \Psi(p, e)$  using  $\mathbb{F}_1$  scores<sup>3</sup>. In particular, we only compared the sub-graph containing the object instances mentioned in  $p$  and  $\hat{p}$ . We describe the details of  $\mathbb{F}_1(\hat{G}, G)$  in the supplementary materials. Inspired by [2], we also compute  $\mathbb{F}_1$ -state and  $\mathbb{F}_1$ -relation.

Furthermore, inspired by program synthesis, we care whether the generated programs are “compilable” as well. We evaluate if the generated programs can be parsed (parsibility) and executed (executability) by the simulators. We will describe the detailed definition of them in the supplementary materials.

### 6.2. Baselines

We implement five different baselines to compare with the proposed *ResActGraph*.

**Nearest Neighbors:** For every example in the testing set, we retrieve the training sample that has a sketch with the highest LCS. In case of a tie, we pick the one with the most similar initial graph.

**Unaries:** We set  $K = 0$  in Eq. 7. This model does not consider the relations of the objects. We use it to showcase the benefits of modeling object relations.

**Graph:** This model does not consider the change of graphs induced by programs (Eq. 8).

**FCActGraph:** This model uses a FC layer to model the graph changes. Specifically, it takes the  $[h_v^{K_{t-1}}, Emb(\alpha_{t-1}), m_{t-1}]$  as inputs and outputs  $h_v^{K_t}$ .

**GRUActGraph:** This model treats the graph changes as another sequence and uses a GRU to ingest  $[Emb(\alpha_{t-1}), m_{t-1}]$  as inputs and considers  $h_v^{K_{t-1}}$  as hidden state to output  $h_v^{K_t}$ .

### 6.3. Results

We show the results of *ResActGraph* quantitatively and qualitatively. Next, we show the ablation study of the number of the graph propagation steps. Finally, we show the prediction results of the whole system.

**Program generation from sketches and graphs.** The results are shown in Table 1. By comparing the Graph and Unaries, we show that aggregating information from neighboring nodes increases performances in nearly all metrics.

The three bottom rows of Table 1 show the results of models that consider graph changes induced by programs. The  $\mathbb{F}_1$  scores and executability benefit the most, which is expected. For example, suppose there is a glass near an opened cabinet in the environment and the model predicts (Grab Glass, Put Glass Cabinet, Close

<sup>3</sup>If the generated programs cannot be parsed or cannot be executed, the  $\mathbb{F}_1$  is set to be 0.

	LCS	$\mathbb{F}_1$ -relation	$\mathbb{F}_1$ -state	$\mathbb{F}_1$	Executability	Parsability
Nearest Neighbors	0.127	0.019	0.288	0.041	-	-
Unaries	0.372	0.16	0.142	0.159	24.8%	75.3%
Graph	0.4	0.171	0.171	0.172	23.1%	82.2%
FCActGraph	0.469	0.261	0.273	0.263	33.7%	<b>88.6%</b>
GRUActGraph	0.508	<b>0.41</b>	0.408	0.411	48.9%	87.9%
ResActGraph	<b>0.516</b>	<b>0.41</b>	<b>0.42</b>	<b>0.413</b>	<b>49.3%</b>	85.3%

Table 1. Induce program from ground truth sketches and ground truth graphs. ( $K = 2$ )

Sketch	GT Program	Generated Program
[Open] <washing machine> [Put] <basket> <washing machine> [Put] <soap> <washing machine> [SwitchOn] <washing machine>	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket> (1000) [Grab] <basket> (1000) [Walk] <bathroom> (1)	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket> (1000) [Grab] <basket> (1000) [Find] <washing machine> (1001) [Open] <washing machine> (1001) [Put] <basket> (1000) <washing machine> (1001)
Environment	[Walk] <washing machine> (1001) [Find] <washing machine> (1001) [Open] <washing machine> (1001) [Put] <basket> (1000) <washing machine> (1001) [Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing machine> (1001) [Find] <washing machine> (1001) [Close] <washing machine> (1001) [PlugIn] <washing machine> (1001) [SwitchOn] <washing machine> (1001)	[Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing machine> (1001) [Close] <washing machine> (1001) [Plugin] <washing machine> (1001) [SwitchOn] <washing machine> (1001)
Washing machine (1001) is closed Washing machine (1001) is off Washing machine (1001) is unplugged Washing machine (1001) in bathroom (1) Soap (1002) inside Washing machine (1001)		

Figure 5. An example of the prediction of ResActGraph. We color the LCS between the prediction and ground truth in light green. Note that the sketch is environment agnostic, so it does not specify the ‘id’ (the number in the parentheses) of the object instances.

	LCS	$\mathbb{F}_1$ -relation	$\mathbb{F}_1$ -state	$\mathbb{F}_1$
Unaries ( $K=0$ )	0.372	0.16	0.142	0.59
ResActGraph ( $K=1$ )	0.427	0.262	0.271	0.264
ResActGraph ( $K=2$ )	<b>0.516</b>	<b>0.41</b>	<b>0.42</b>	<b>0.413</b>
ResActGraph ( $K=3$ )	0.513	0.399	0.407	0.401

Table 2. Ablation study of the propagation steps  $K$ .

Cabinet) at the first three steps, if the model wants to grab other things from the cabinet without opening it after  $t \geq 4$ , it fails since the cabinet is closed at  $t = 3$ .

Among the three bottom rows of Table 1, the proposed ResActGraph performs the best in  $\mathbb{F}_1$ . The reason is that using the residual architecture is easier for the model to learn the state “changes” compared to using FC. Using GRU to encode the state changes is also an alternative, but we observe that it converges slower since it has more number of parameters to learn and does not perform better.

In Fig. 5, we show the qualitative results of ResActGraph. Even though the generated program does not exactly match the ground truth, it reaches nearly the same environment state. In Fig. 6, we show the results with the same sketch, but different initial environment states. Note that we

only show the states and relations related to the programs. The model correctly induces correct actions w.r.t. the environment changes and the two generated programs nearly reach the same environment states.

**Ablation studies.** We show the effect of the number of propagation steps  $K$  in Table 2. Both the baseline and the proposed model benefit as  $K$  increases, and the proposed model performs better than the baseline regardless of different  $K$ . We found that the performance saturates when  $K = 2$ , so we fixed it for all other experiments.

**Combining predicted sketches with ResActGraph.** The LCS of  $f_{demo2sketch}$  and  $f_{desc2sketch}$  are 0.15 and 0.27 respectively. The reason why the LCS is low is that they are significantly shorter (on average 2.4 instructions) than the programs (on average 18.79). This makes the sketch prediction a quite challenging task where LCS is highly penalized even under small errors.

With the trained model, we can directly generate the programs from the demonstrations or descriptions. Note that we do not re-train the program generation model. In Table 3 and Table 4, we show the results of the program generation with the sketches predicted from descriptions and demonstrations respectively. The performance gap between

Sketch		[Sit] <sofa> [Read] <book>	
Environment 1	Generated Program 1	Environment 2	Generated Program 2
Book (263) in bedroom (23) Bookmark (27) near book (263) Bookmark (27) in bedroom (23) Sofa (101) in bedroom (23) Sofa (101) near bookmark (275) Sofa (101) is free Book (263) near sofa (1001)	[Walk] <bedroom> (23) [Walk] <book> (263) [Find] <book> (263) [Grab] <book> (263) [Find] <sofa> (101) [Sit] <sofa> (101) [Find] <bookmark> (27) [Read] <book> (263)	Sofa (101) in bedroom (23) Book (263) in bedroom (23) Bookmark (27) in bedroom (23) Sofa (101) occupied Phone (75) on sofa (101) Cards (261) on sofa (101) Game (231) on sofa (101) Check (213) on sofa (101)	[Walk] <bedroom> (23) [Walk] <sofa> (101) [Find] <sofa> (101) [Find] <phone> (75) [Grab] <phone> (75) [Release] <phone> (75) [Find] <cards> (261) [Grab] <cards> (261) [Release] <cards> (261) [Find] <game> (231) [Grab] <game> (231) [Release] <game> (231) [Find] <check> (213) [Grab] <check> (213) [Release] <check> (213) [Sit] <sofa> (101) [Standup] [Walk] <book> (263) [Find] <book> (263) [Grab] <book> (263) [Read] <book> (263)

Figure 6. An example of the prediction of ResActGraph with the same sketch, but different environments. We highlight the difference between two environments with orange and color the LCS between two predictions in light green.


Demonstration	Environment	Generated Sketch	Generated Program	
	Phone (10) in living room (1)	[Grab] <phone>	[Walk] <living room> (1) [Walk] <phone> (10) [Find] <phone> (10)	[Walk] <phone> (10) [TurnTo] <phone> (10) [LookAt] <phone> (10) [Grab] <phone> (10)
Description	Environment	Generated Sketch	Generated Program	
Walk into the home office. Walk up to the chair, sit down in the chair. Type with the keyboard.	Chair (29) close to keyboard (2) Chair (29) is free Keyboard (2) close to computer Computer (31) in living room (1)	[Sit] <chair> [Type] <keyboard>	[Walk] <living room> (1) [Walk] <desk> (137) [Find] <chair> (29)	[Sit] <chair> (29) [Find] <keyboard> (2) [Type] <keyboard> (2)

Figure 7. Predictions from the ResActGraph given sketches from descriptions and demonstrations.

	LCS	$F_1$ -relation	$F_1$ -state	$F_1$	Executability	Parsability
Unaries	0.289	0.188	0.191	0.189	44.4%	73.4%
Graph	0.297	0.24	0.233	0.241	43.9%	89.9%
ResActGraph	<b>0.331</b>	<b>0.347</b>	<b>0.339</b>	<b>0.348</b>	<b>63.3%</b>	<b>92.5%</b>

Table 3. Induce program from sketches predicted from descriptions and ground truth graphs. ( $K = 2$ )

	LCS	$F_1$ -relation	$F_1$ -state	$F_1$	Executability	Parsability
Unaries	0.257	0.099	0.09	0.098	25.6%	74%
Graph	0.284	0.202	0.202	0.203	34.3%	82.8%
ResActGraph	<b>0.327</b>	<b>0.316</b>	<b>0.323</b>	<b>0.318</b>	<b>54.7%</b>	<b>86%</b>

Table 4. Inducing program from sketches predicted from demonstrations and ground truth graphs. ( $K = 2$ )

the proposed model and the baselines becomes small. The reason is that the model is confused when the non-perfect sketches are given, resulting in similar performance. Note that all models still perform better than Unaries. Qualitative results are shown in Fig. 7, showing that the model predicts the plausible sketch and ResActGraph generates plausible programs w.r.t to the sketch and the graph.

## 7. Conclusion

In this work, we propose the *environment-aware program generation* task. We introduce sketches as environment-independent activity representations and address the problem in two steps: generating sketches from demonstrations or descriptions and generating programs

from sketches and graphs. To this end, we propose a novel model, *ResActGraph* and create a dataset *VirtualHome-Env*, with sketches, environments, and programs.

The *environment-aware program generation* is far from being solved and opens exciting research directions. While we access the truth state of the environment graph, one natural extension would be to predict it from environment observations [21], but this would still require an oracle providing images of the interior of closed objects or unexplored areas. Additionally, though the *ResActGraph* updates the hidden states of nodes at each time step, the graph structure is not changed, which impedes from performing message passing at each time step. Finally, our setting assumes a fully observable environment. Working with partially observable environments opens up exciting directions, such as dealing with environment changes due to factors external to the agent or modelling the agent’s theory of mind.

**Acknowledgement** The authors would like to thank Chaoqi Wang, Tsun-Hsuan Wang, and Meng-Li Shih for helpful discussions and feedback for the early draft. X.P. is supported by La Caixa Fellowship.



## References

- [1] M. Alomari, P. Duckworth, M. Hawasly, D. C. Hogg, and A. G. Cohn. Natural language grounding and grammar induction for robotic manipulation commands. In *Proceedings of the First Workshop on Language Grounding for Robotics*, pages 35–43, 2017. 2
- [2] P. Anderson, B. Fernando, M. Johnson, and S. Gould. Spice: Semantic propositional image caption evaluation. In *European Conference on Computer Vision*, pages 382–398. Springer, 2016. 6
- [3] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth. Robotic roommates making pancakes. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 529–536. IEEE, 2011. 2
- [4] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016. 2
- [5] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014. 4
- [6] L. Dong and M. Lapata. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018. 2
- [7] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine. One-shot visual imitation learning via meta-learning. In *Conference on Robot Learning*, pages 357–368, 2017. 2
- [8] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016. 2
- [9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 10
- [10] D.-A. Huang, S. Nair, D. Xu, Y. Zhu, A. Garg, L. Fei-Fei, S. Savarese, and J. C. Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration, 2018. 2
- [11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 10
- [12] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. 3
- [13] H. Mei, M. Bansal, and M. R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences, 2015. 2
- [14] J. Merel, Y. Tassa, S. Srinivasan, J. Lemmon, Z. Wang, G. Wayne, and N. Heess. Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*, 2017. 2
- [15] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017. 1, 2
- [16] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine. Combining self-supervised learning and imitation for vision-based rope manipulation. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2146–2153. IEEE, 2017. 2
- [17] X. Puig, K. Ra, M. Boben, J. Li, T. Wang, S. Fidler, and A. Torralba. Virtualhome: Simulating household activities via programs. *arXiv preprint arXiv:1806.07011*, 2018. 1, 2
- [18] A. Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008. 1, 2, 5
- [19] S.-H. Sun, H. Noh, S. Somasundaram, and J. Lim. Neural program synthesis from diverse demonstration videos. In *Proceedings of the 35th International Conference on Machine Learning*, 2018. 2
- [20] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI’11, pages 1507–1514. AAAI Press, 2011. 2
- [21] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5410–5419, 2017. 8
- [22] Y. Zhu, D. Gordon, E. Kolve, D. Fox, L. Fei-Fei, A. Gupta, R. Mottaghi, and A. Farhadi. Visual semantic planning using deep successor representations. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 483–492, 2017. 2

	Conv_1+BN	Conv_2+BN	Conv_3+BN	FC
Input	Kernel: 5*5	Kernel: 5*5	Kernel: 5*5	
(size: 240, 320, 3)	Stride: 1	Stride: 1	Stride: 3	256
	Padding: 2	Padding: 2	Padding: 2	
	Channel: 16	Channel: 32	Channel: 32	

Table 5. The model architecture of the CNN that ingests the semantic segmentation map.

	<i>VirtualHome</i>	Executable	Augmented	<i>VirtualHome-Env</i>
# programs	2807	1387	27284	32761
Average length	13.26	9.61	18.58	18.79
Average # nodes	-	289	290	289
Average # edges	-	4393	4305	4319

Table 6. The statistics of *VirtualHome-Env*. The fact that some programs in *VirtualHome* are noisy results in less number of executable programs.

## 8. Appendices

### 8.1. Implementation details

To train ResActGraph, we use the ground truth graphs extracted from Unity to represent the environments. We set the propagation time  $K = 2$  except for the ablation studies. To train the model that ingests demonstrations to generate sketches, we use a finetuned Resnet-18 [9] to extract image features and a 3-layer CNN to extract features from semantic segmentation map. All the GRUs are with 256 hidden units except for those in GGNN are with 32.

The following is the details of the model ResActGraph, Desc2sketch, and Demo2sketch:

**RNN-ResActGraph.** The hidden units of GRU is 256 and the hidden units of GGNN is 32. We use Adam [11] with  $3 * 10^{-4}$  as initial learning rate and mini-batch size 32. Schedule sampling is applied to mitigate the exposure bias. We use linear schedule with max probability 1.0 and min probability 0.3 and decrease in 20000 steps.

**Desc2sketch.** We use seq2seq model with GRU. The hidden units of GRU is 256. We use Adam with  $3 * 10^{-4}$  as initial learning rate and mini-batch size 64. Schedule sampling is applied to mitigate the exposure bias. We use linear schedule with max probability 1. and min probability 0. and decrease in 20000 steps.

**Demo2sketch.** We finetune Resnet-18 to extract image features and build another CNN to ingest the ground truth semantic segmentation maps. The CNN structure is shown in Table. 5. On the other hand, we use GRU to decode the sketches. The hidden units of GRU is 256. We use Adam with  $3 * 10^{-4}$  as initial learning rate and mini-batch size 16. Schedule sampling is applied to mitigate the exposure bias. We use linear schedule with max probability 1. and min probability 0. and decrease in 10000 steps.

### 8.2. Dataset statistics

We show in statistics of the collected dataset in table 6

### 8.3. Method for dataset augmentation

We formally describe how we augment the dataset of programs using exceptions. Let  $p$  be the original program and  $e'$  the modified environment. Let  $\Gamma_\zeta$  be the function that for some exception  $\theta$ , takes a program  $p$  and environment  $e'$  corrects the program into  $p'$ . Finally, let  $\Omega$  be the function that takes  $p, e$  as the input and outputs its post-conditions<sup>4</sup>. We formally define the program extension as:

<sup>4</sup>A post-condition is a condition or predicate that must always be true just after the execution of some section of code.

$$\begin{aligned}
 p' &= \Gamma_\zeta(p, e'), \\
 e_{final} &= \Psi(e', p') \\
 &s.t. \forall j \in \Omega(p), e_{final} \text{ satisfies } j
 \end{aligned} \tag{11}$$

The algorithm is shown in 1:

### 8.4. Details of the Evaluation Metrics

We compute the  $\mathbb{F}_1(\hat{G}, G)$  as follows:

$$\begin{aligned}
 \mathbb{P}(\hat{G}, G) &= \frac{\|\mathbb{T}(\hat{G})\| \otimes \|\mathbb{T}(G)\|}{\|\mathbb{T}(\hat{G})\|} \\
 \mathbb{R}(\hat{G}, G) &= \frac{\|\mathbb{T}(\hat{G})\| \otimes \|\mathbb{T}(G)\|}{\|\mathbb{T}(G)\|} \\
 \mathbb{F}_1(\hat{G}, G) &= \frac{2 \cdot \mathbb{P}(\hat{G}, G) \cdot \mathbb{R}(\hat{G}, G)}{\mathbb{P}(\hat{G}, G) + \mathbb{R}(\hat{G}, G)}
 \end{aligned}$$

where  $\otimes$ ,  $\mathbb{P}$ , and  $\mathbb{R}$  are the binary matching function, precision and recall respectively.  $\mathbb{T}$  is a function that converts graphs to tuples.

Parsability refers to the syntactic correctness of the programs, evaluating whether the number of object arguments given the action is valid. Sleep TV is invalid, since Sleep does not expect any object. Executability refers to the semantic correctness, meaning that 1) the combination of objects and actions is valid (e.g., Grab Bedroom is invalid) and 2) the instruction can be performed under a given environment state (e.g., Grab Mug is only valid if the mug is close).

### 8.5. Further analysis of the experiments

The performance of the sketch prediction from descriptions is far better than from demonstrations (shown in Sec. 6.3). We find that only 7% of the testing vase are more easily predicted from demonstrations than descriptions, especially for the activities pet cat and pick up phone. One potential

---

**Algorithm 1** Pseudo-code of the method described in Sec. 5.3. Given a program  $p$  with preconditions  $\Phi(p)$ , the procedure produces at most  $n$  new programs.

---

```

1: procedure AUGMENT( $p, \Phi(p), n$ )
2:   perturbed-set  $\leftarrow$  set()
3:   while len(perturbed-set) <  $n$  do ▷ Perturb preconditions
4:      $z \leftarrow$  rand(0, 1)
5:      $\Phi(p)' \leftarrow$  perturb( $\Phi(p), z$ )
6:      $e' \sim E_{\Phi(p)'}$  ▷ Sample with eq. 10
7:     perturbed-set.add( $e'$ )
8:   end while
9:   final-progs  $\leftarrow$  list()
10:   $p' \leftarrow p$ 
11:  foreach  $e' \leftarrow$  perturbed-set do
12:    iter  $\leftarrow$  1
13:    valid  $\leftarrow$  False
14:    while not valid and iter < maxiter do
15:      try
16:         $\Psi(p', e')$  ▷ Execute in simulator
17:        final-progs.add( $p'$ )
18:        valid  $\leftarrow$  True
19:      catch ExceptionType
20:         $p' \leftarrow \Gamma_{\text{ExceptionType}}(p', e')$  ▷ Solve with eq. 11
21:        iter  $\leftarrow$  iter + 1
22:      end try
23:    end while
24:  end foreach
25:  return final-progs
26: end procedure

```

---

reason is that the average length of the descriptions of these activities (15.3 words) is longer than those of other activities (13.4).

## 8.6. Knowledge base

### 8.6.1 RealEnv

We collected two knowledge bases that aim to set up our environments. The first one, *ObjectStates KB*, contains information about which states correspond to which objects. We collected 29 different states and 9 of them are considered by the simulator. Table 8 shows the collected states with an indication of whether they are included in the simulator.

The second knowledge base, *ObjectRelations KB*, contains typical spatial relations between objects that aims at obtaining realistic environment layouts to set up the scene for each activity. We show a few examples of spatial relations in Table 7. In total, it contains 505 types of spatial relations included.

### 8.6.2 ExceptionHandler

We show the exceptions and subroutines used to augment the dataset in Table 9. Each row shows the name of the exception, the case when the exception fires and a description of the subroutine that is called to correct the given exception.

## 8.7. Additional Qualitative Results

We now show additional qualitative results of our *RNN-ResActGraph*. In Fig. 8 and Fig. 9, we use the ground truth sketches as the input to the model. Although the generated programs are not exactly the same as the ground truth programs, the generated program does achieve similar environment state as the environment state of ground truth.

In Fig. 10, we show how the generated program changes when using the same sketches but different graphs as input. We highlight the difference between two environments with orange color. From both cases, we observe that the generated programs achieve almost the same environment state even though they start from different environment states.

In Fig. 11, we show the generated programs from the sketches predicted from descriptions. It shows that sometimes our model cannot generate the exact same program as ground truth, but the generated program does achieve similar environment state as the environment state of the ground truth. This is also shown in the main paper where the  $F_1$  scores of the proposed model are higher than others.

Lastly, we show one example of the of our simulators in Fig. 12

States	Examples	In simulator
On/Off	Television, Computer, Microwave, Oven, Stereo	✓
Plugged/Unplugged	Television, Computer, Microwave, Oven, Stereo	✓
Open/Closed	Fridge, Cabinet, Folder, Bag, Purse, Blender	✓
Free/Occupied	Sofa, Table, Sink, Toilet, Piano, Shoe Rack, Bed	✓
Dirty/Clean	Apple, Floor, Bowl, Socks, Plate, Cupboard	
Full/Empty	Washing machine, Dishwasher, Measuring Cup	
Grabbed	Apple, Towel, Book, Glass, Plate, Toy, Cat, Pot	✓
Flushed	Toilet	
Sharpened	Knife	
Wet/Dry	Towel, Plate, Glass, Napkin, Shirt, Toothbrush	
Peeled/Sliced/Squeezed	Banana, Vegetable, Fish, Fruit, Cake	
Burning	Plate, Pot, Pan, Fish, Water, Candle, Chicken	
Frozen	Water, Vegetable, Ice cream, Wine, Hamburger	
Folded/Unfolded	Sheets, Shirt, Jacket, Ironing Board, Placemat	
Hot/Cold	Plate, Bowl, Water, Hamburger, Oven, Microwave	
Cooked/Raw/Rotten	Egg, Fish, Vegetable, Meat, Fruit, Pasta, Pizza	

Table 7. ObjectStates KB

Object	In relations	On relations	Nearby Relations
Wine Glass	Cupboard, Cabinet, Dishwasher	Coffee table, Dishrack, Kitchen Counter	
Milk	Fridge, Microwave, Trashcan, Sauce pan	Kitchen counter, Table, Pantry	
Remote Control	Dresser	Sofa, Love Seat, Bed, Table, Mousepad	Dvd player
Pot	Microwave, Cupboard, Cabinet	Dishrack, Kitchen Counter, Stove	
Razor	Cabinet, Trashcan, Bag, Box	Table, Bathroom counter	
Cd Player	Closet, Cabinet, Bag	Table, Couch	Stereo
Chicken	Fridge, Trashcan, Freezer, Sauce Pan, Oven	Kitchen Counter, Cutting Board, Plate	
Oven mitts	Oven, Cabinet, Basket for clothes	Table, Kitchen counter	
Document	Folderm Garbage Can, Cabinet, Filing Cabinet	Desk, Sofa, Bookshelf, Music Stand	
Sheets	Basket for clothes, Dresser, Washing Machine, Cabinet	Couch, Love Seat, Bed	

Table 8. ObjectRelations KB



Exception name	When is raised	Subroutine
IS_OPEN	Character aims to open something that is already open	Remove the <i>Open</i> instruction
IS_CLOSED	Character aims to close something that is closed	Remove the <i>Close</i> instruction
STANDING	1) Character aims to stand up but is not sitting, 2) Character aims to sleep but is not lying nor sitting	1) remove <i>Stand Up</i> , 2) add instruction <i>Sit</i> or <i>Lie</i>
SITTING	1) Character aims to <i>Sit</i> but is sitting, 2) Character aims to walk but is sitting	1) Remove instruction <i>Sit</i> , 2) Add instruction <i>Stand up</i>
NOT_CLOSE	Character wants to interact with an object but it is not close enough	Add instruction <i>Walk towards the object</i>
NOT_FACING	Character wants to <i>Watch</i> and object (e.g. TV) but is not facing it.	Add instruction <i>textTurnTo</i> the object
IS_ON	1) Character switches on an object that is already on 2) Character aims to open a utility (e.g. washing machine) before turning it off	1) Remove instruction <i>Turn On</i> 2) Insert an instruction <i>Switch Off</i> .
IS_OFF	1) Character switches off an object that is off, 2) character watches TV but it is off	1) Remove instruction <i>Switch Off</i> , 2) Insert <i>Switch On</i>
PLUGGED	1) Character plugs in an object that is already plugged	Remove instruction <i>PlugIn</i>
UNPLUGGED	1) Character plugs out an object that is plugged out 2) Character switches on an object that is plugged out	1) Remove instruction <i>PlugOut</i> 2) Insert instruction <i>PlugIn</i>
OCCUPIED	There are objects in a surface where the character aims to sit or lie	Obtain all the objects in the occupied surface and insert <i>Grab</i> and <i>Release</i> instructions to remove them
INSIDE_CLOSED	The character aims to interact an object that is inside some closed one (e.g. a fridge)	Obtain the object containing the target object, and open it
FREE_HAND	The Character wants to interact with an object but both of their hands are occupied	Leave one object from the hands, Interact with object and grab object again

Table 9. Exceptions handled to augment the programs dataset, based on perturbation of the environment

Sketch	GT Program	Generated Program
[Grab] <tooth paste> [Put] <tooth paste> <toothbrush holder> [Switchon] <faucet> [Rinse] <toothbrush>	[Walk] <bathroom> (32) [Walk] <sink> (9) [Find] <toothbrush holder> (63) [Turnto] <toothbrush holder> (63) [Lookat] <toothbrush holder> (63) [Find] <tooth paste> (15) [Grab] <tooth paste> (15) [Find] <toothbrush> (2) [Grab] <toothbrush> (2) [Walk] <toothbrush holder> (63) [Put] <tooth paste> (15) <toothbrush holder> (63) [Find] <faucet> (70) [Rinse] <toothbrush> (2) [Walk] <toothbrush holder> (63) [Put] <toothbrush> (2) <toothbrush holder> (63)	[Walk] <bathroom> (32) [Walk] <sink> (9) [Find] <toothbrush holder> (63) [Turnto] <toothbrush holder> (63) [Lookat] <toothbrush holder> (63) [Find] <tooth paste> (15) [Grab] <tooth paste> (15) [Find] <toothbrush> (53) [Grab] <toothbrush> (53) [Walk] <toothbrush holder> (63) [Put] <tooth paste> (15) <toothbrush holder> (63) [Rinse] <toothbrush> (2) [Walk] <faucet> (70) [Switchoff] <faucet> (70)
<b>Environment</b>		
Faucet (70) near Sink (9) Faucet (70) is on Faucet (70) in Bathroom (32) Tooth paste (15) near Sink (9) Toothbrush holder (63) in Sink (9) Toothbrush (2) in Bathroom (32)		

Sketch	GT Program	Generated Program
[Wash] <hands> [Grab] <soap> [Rinse] <hands> [Wipe] <hands>	[Walk] <bathroom> (41) [Walk] <sink> (93) [Find] <sink> (93) [Turnto] <sink> (93) [Find] <faucet> (10) [Switchon] <faucet> (10) [Find] <hands> (92) [Wash] <hands> (92) [Find] <soap> (77) [Grab] <soap> (77) [Scrub] <hands> (92) [Putobjback] <soap> (77) [Rinse] <hands> (92) [Switchoff] <faucet> (10) [Find] <towel> (8) [Grab] <towel> (8) [Wipe] <hands> (92) [Putobjback] <towel> (8)	[Walk] <bathroom> (41) [Walk] <sink> (93) [Find] <sink> (93) [Turnto] <sink> (93) [Find] <faucet> (10) [Switchon] <faucet> (10) [Find] <hands> (92) [Rinse] <hands> (92) [Find] <soap> (77) [Grab] <soap> (77) [Squeeze] <soap> (77) [Wash] <hands> (92) [Rinse] <hands> (92) [Switchoff] <faucet> (10) [Find] <towel> (8) [Grab] <towel> (8) [Wipe] <hands> (92) [Putobjback] <towel> (8)
<b>Environment</b>		
Faucet (10) is off Faucet (10) near Sink (93) Faucet (10) in Bathroom (41) Soap (77) in Bathroom (41) Towel (8) in Bathroom (41) Hands (92) near Sink (93)		

Sketch	GT Program	Generated Program
[Grab] <paper towel> [Wipe] <hands>	[Walk] <bathroom> (81) [Walk] <paper towel> (54) [Find] <paper towel> (54) [Turnto] <paper towel> (54) [Lookat] <paper towel> (54) [Grab] <paper towel> (54) [Find] <hands> (4) [Wipe] <hands> (4)	[Walk] <bathroom> (81) [Walk] <paper towel> (54) [Find] <paper towel> (54) [Grab] <paper towel> (54) [Find] <hands> (4) [Wipe] <hands> (4) [Putobjback] <paper towel> (54)
<b>Environment</b>		
Hands (4) near Paper Towel (54) Paper Towel (54) in Bathroom (81)		

Figure 8. Predictions from the RNN-ResActGraph from the GT sketch and the environment, along with the GT programs. We color the longest common subsequence in light green.

Sketch	GT Program	Generated Program
[Putoff] <clothes jacket> [Put] <clothes jacket> <hanger>	[Walk] <bedroom> (28) [Walk] <clothes jacket> (19) [Find] <clothes jacket> (19) [Find] <clothes jacket> (19) [Putoff] <clothes jacket> (19) [Grab] <clothes jacket> (19) [Find] <hanger> (18) [Put] <clothes jacket> (19) <hanger> (18)	[Walk] <bedroom> (28) [Walk] <clothes jacket> (19) [Find] <clothes jacket> (19) [Find] <clothes jacket> (19) [Putoff] <clothes jacket> (16) [Grab] <clothes jacket> (16) [Find] <bed> (4) [Put] <clothes jacket> (19) <bed> (4)
<b>Environment</b>		
Hanger (18) near Clothes Jacket (19) Hanger (18) in Bedroom (28) Bed (4) in Bedroom (28) Clothes Jacket (19) in Character		
Sketch	GT Program	Generated Program
[Lie] <bed> [Sleep]	[Walk] <bedroom> (6) [Walk] <lamp> (72) [Switchoff] <lamp> (72) [Walk] <bed> (79) [Lie] <bed> (79) [Sleep]	[Walk] <bedroom> (6) [Walk] <bed> (79) [Find] <lamp> (72) [Lie] <bed> (79) [Sleep]
<b>Environment</b>		
Lamp (72) is on Lamp (72) near Bed (79) Bed (79) in Bedroom (6) Bed (79) is free		
Sketch	GT Program	Generated Program
[Greet] <child> [Greet] <man> [Greet] <woman>	[Walk] <living room> (12) [Walk] <child> (67) [Find] <child> (67) [Greet] <child> (67) [Find] <man> (48) [Greet] <man> (48) [Find] <woman> (59) [Greet] <woman> (59)	[Walk] <living room> (12) [Walk] <child> (67) [Find] <child> (67) [Greet] <child> (67) [Find] <woman> (59) [Greet] <woman> (59) [Find] <man> (48) [Greet] <man> (48)
<b>Environment</b>		
Man (48) near Child (67) Woman (59) near Child (67) Man (48) in Living Room (12)		
Sketch	GT Program	Generated Program
[Switchon] <faucet> [Put] <glass> <sink> [Drink] <glass>	[Walk] <dining room> (72) [Walk] <sink> (22) [Find] <glass> (99) [Grab] <glass> (99) [Find] <faucet> (15) [Switchon] <faucet> (15) [Find] <water> (5) [Put] <glass> (99) <sink> (22) [Grab] <glass> (99) [Switchoff] <faucet> (15) [Drink] <glass> (99)	[Walk] <dining room> (72) [Walk] <glass> (99) [Find] <glass> (99) [Grab] <glass> (99) [Find] <faucet> (15) [Switchon] <faucet> (15) [Find] <water> (5) [Find] <sink> (22) [Put] <glass> (99) <sink> (22) [Grab] <glass> (99) [Switchoff] <faucet> (15) [Drink] <glass> (99)
<b>Environment</b>		
Faucet (13) near Sink (68) Faucet (13) in Dining Room (72) Water (8) near Sink (68) Glass (99) in Sink (68)		

Figure 9. Predictions from the RNN-ResActGraph from the GT sketch and the environment, along with the GT programs. We color the longest common subsequence in light green.

Sketch		[open] <washing_machine> [putback] <basket> <washing_machine> [putback] <soap> <washing_machine> [switchon] <washing_machine>	
Environment 1	Generated Program 1	Environment 2	Generated Program 2
Soap (1002) near washing machine (1001) Soap (1002) inside washing machine (1001) Washing machine (1001) is opened Washing machine (1001) is on Washing machine (1001) is plugged	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket> (1000) [Grab] <basket> (1000) [Find] <washing_machine> (1001) [SwitchOff] <washing_machine> (1001) [Open] <washing_machine> (1001) [Put] <basket> (1000) <washing_machine> (1001) [Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing_machine> (1001) [Close] <washing_machine> (1001) [SwitchOn] <washing_machine> (1001)	Soap (1002) near washing machine (1001) Washing machine (1001) is closed Washing machine (1001) is off Washing machine (1001) is unplugged	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket> (1000) [Grab] <basket> (1000) [Find] <washing_machine> (1001) [Open] <washing_machine> (1001) [Put] <basket> (1000) <washing_machine> (1001) [Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing_machine> (1001) [Close] <washing_machine> (1001) [Plugin] <washing_machine> (1001) [SwitchOn] <washing_machine> (1001)

Sketch		[putback] <groceries> <fridge>	
Environment 1	Generated Program 1	Environment 2	Generated Program 2
Grocery (1001) near fridge (1000) Fridge (1000) in dining room (47) Grocery (1001) in dining room (47) Fridge (1000) is closed	[Walk] <dining_room> (47) [Walk] <fridge> (1000) [Find] <fridge> (1000) [Open] <fridge> (1000) [Find] <groceries> (1001) [Grab] <groceries> (1001) [Put] <groceries> (1001) <fridge> (1000) [Close] <fridge> (1000)	Grocery (1001) near fridge (1000) Fridge (1000) in dining room (47) Grocery (1001) in dining room (47) Fridge (1000) is opened	[Walk] <dining_room> (47) [Walk] <fridge> (1000) [Find] <fridge> (1000) [Find] <groceries> (1001) [Grab] <groceries> (1001) [Put] <groceries> (1001) <fridge> (1000) [Close] <fridge> (1000)

Figure 10. Predictions from the RNN-ResActGraph model from the same sketches but different environments. We color the difference between environments in orange and the longest common subsequence in light green.



Description	pick up book. sit on sofa. open book. read	Description	turn on reading light. bring book to couch. sit on couch. <unk> pillow. put pillow <unk> my back. read my book.
Sketch	[sit] <chair>, [read] <book>	Sketch	[sit] <sofa>, [read] <book>
GT Program	[walk] <living_room> (21) [walk] <book> (1001) [find] <book> (1001) [grab] <book> (1001) [find] <chair> (1002) [sit] <chair> (1002) [read] <book> (1001)	GT Program	[walk] <bedroom> (276) [walk] <book> (1001) [find] <book> (1001) [grab] <book> (1001) [find] <sofa> (201) [sit] <sofa> (201) [standup] [walk] <bookmark> (1004) [find] <bookmark> (1004) [read] <book> (1001)
Generated Program	[walk] <living_room> (21) [walk] <book> (1001) [find] <book> (1001) [grab] <book> (1001) [find] <chair> (1002) [sit] <chair> (1002) [read] <book> (1001)	Generated Program	[walk] <bedroom> (276) [walk] <book> (1001) [find] <book> (1001) [grab] <book> (1001) [find] <sofa> (201) [sit] <sofa> (201) [read] <book> (1001)

Figure 11. Predictions from the RNN-ResActGraph from the predicted sketch (from descriptions) and the environment, along with the GT programs. We color the longest common subsequence in light green.



Figure 12. An example of snapshots generated in VirtualHome for a given program.