CraftAssist Instruction Parsing: Semantic Parsing for a Voxel-world Assistant

Kavya Srinet[*]
Facebook AI Research
ksrinet@fb.com

Yacine Jernite[*†]
HuggingFace
yacine@huggingface.co

Jonathan Gray
Facebook AI Research
jsgray@fb.com

Arthur Szlam
Facebook AI Research
aszlam@fb.com

Abstract

We propose a semantic parsing dataset focused on instruction-driven communication with an agent in the game Minecraft[1]. The dataset consists of 7K human utterances and their corresponding parses. Given proper world state, the parses can be interpreted and executed in game. We report the performance of baseline models, and analyze their successes and failures.

1 Introduction

Semantic parsing is used as a component for natural language understanding in human-robot interaction systems (Lauria et al., 2001; Bos and Oka, 2007; Tellex et al., 2011; Matuszek et al., 2013; Thomason et al., 2019), and for virtual assistants (Campagna et al., 2017; Kollar et al., 2018; Campagna et al., 2019). We would like to be able to apply deep learning methods in this space, as recently researchers have shown success with these methods for semantic parsing more generally, e.g. (Dong and Lapata, 2016; Jia and Liang, 2016; Zhong et al., 2017). However, to fully utilize powerful neural network approaches, it is necessary to have large numbers of training examples. In the space of human-robot (or human-assistant) interaction, the publicly available semantic parsing datasets are small. Furthermore, it can be difficult to reproduce the end-to-end results (from utterance to action in the environment) because of the wide variety of robot setups and proprietary nature of personal assistants.

In this work, we introduce a new semantic parsing dataset for human-bot interactions. Our "robot" or "assistant" is embodied in the sandbox construction game Minecraft[2], a popular multiplayer open-world voxel-based crafting game. We also provide the associated platform for executing the logical forms in game.

Situating the assistant in Minecraft has several benefits for studying task oriented natural language understanding (NLU). Compared to physical robots, Minecraft allows less technical overhead irrelevant to NLU, such as difficulties with hardware and large scale data collection. On the other hand, our bot has all the basic in-game capabilities of a player, including movement and placing or removing voxels. Thus Minecraft preserves many of the NLU elements of physical robots, such as discussions of navigation and spatial object reference.

Working in Minecraft may enable large scale human interaction because of its large player base, in the tens of millions. Furthermore, although Minecraft's simulation of physics is simplified, the task space is complex. While there are many atomic objects in the game, such as animals and block-types, that require no perceptual modeling, the player also interacts with complex structures made up of collections of voxels such as a "house" or a "hill". The assistant cannot apprehend them without a perceptual system, creating an ideal test bed for researchers interested in the interactions between perception and language.

Our contributions in the paper are as follows:
Grammar: We develop a grammar over a set of primitives that comprise a mid-level interface to Minecraft for machine learning agents.
Data: We collect 7K crowd-sourced annotations of commands generated independent of our grammar. In addition to the natural language commands and the associated logical forms, we release the tools used to collect these, which allow

[*]equal contribution
[†]Work done while at Facebook AI Research
[1]Minecraft features: ©Mojang Synergies AB included courtesy of Mojang AB
[2]https://minecraft.net/en-us/. We limit ourselves to creative mode for this work

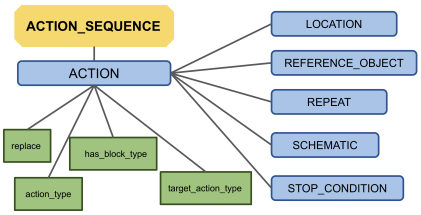

Figure 1: The basic structure of the ACTION_SEQUENCE branch of the assistant's grammar. The gold octagon is an internal node whose children are ordered, blue rectangles are regular internal nodes, and green rectangles are categorical leaf nodes. Not all combinations of children of ACTION are possible, see the full list of possible productions (and the productions for PUT_MEMORY and GET_MEMORY) in the Appendix.

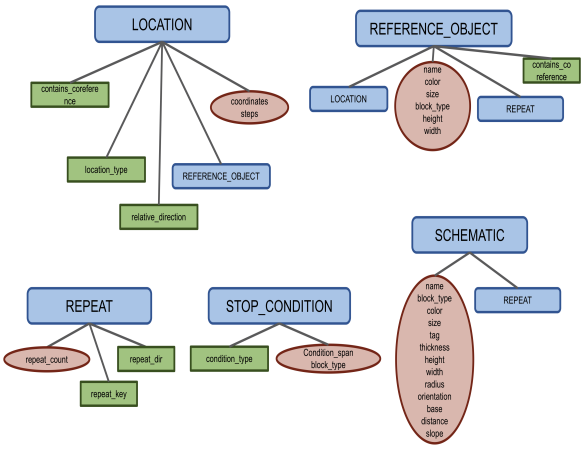

Figure 2: The basic structure of internal nodes in the assistant's grammar. Blue rectangles are internal nodes, green rectangles are categorical leaf nodes, and red ovals are span nodes.

crowd-workers to efficiently and accurately annotate parses.

Models: We show the results of several neural semantic parsing models trained on our data.

Execution: Finally, we also make available the code to execute logical forms in the game, allowing the reproduction of end-to-end results. This also opens the door to using the data for reinforcement and imitation learning with language. We also provide access to an interactive bot using these models for parsing[3].

2 The Assistant Grammar

In this section we summarize a grammar for generating logical forms that can be interpreted into programs for the agent architecture described in (Gray et al., 2019).

2.1 Agent Action Space

The assistant's basic functions include moving, and placing and destroying blocks. Supporting these basic functions are methods for control flow and memory manipulation.

Basic action commands: The assistant can MOVE to a specified location; or DANCE with a specified sequence of steps. It can BUILD an object from a known schematic (or by making a copy of a block-object in the world) at a given location, or DESTROY an existing object. It can DIG a hole of a given shape at a specified location, or FILL one up. The agent can also be asked to complete a partially built structure however it sees fit by FREEBUILD. Finally, it can SPAWN a mob (an animate NPC in Minecraft).

Control commands: Additionally, the agent can STOP or RESUME an action, or UNDO the result of a recent command. Furthermore, the assistant can LOOP given a task and a stop-condition. Finally, it needs to be able to understand when a sentence does not correspond to any of the above mentioned actions, and map it to a NOOP.

Memory interface: Finally, the assistant can interact with its SQL based memory. It can place or update rows or cells, for example for tagging objects. This can be considered a basic version of the self-improvement capabilities in (Kollar et al., 2013; Thomason et al., 2015; Wang et al., 2016, 2017). It can retrieve information for question answering similar to the VQA in (Yi et al., 2018).

2.2 Logical Forms

The focus of this paper is an intermediate representation that allows natural language to be interpreted into programs over the basic actions from the previous section. The logical forms (represented as trees) making up this representation consist of three basic types of nodes: "internal nodes" that can have children, "categorical" (leaf) nodes that belong to a fixed set of possibilities, and "span" nodes that point to a region of text in the natural language utterance. The full grammar is shown in the Appendix; and a partial schematic representation is shown in Figures 1 and 2. In the paragraphs below, we give more detail about some of the kinds of nodes in the grammar.

[3]Instructions can be found at http://craftassist.io/acl2020demo, requires a Minecraft license and client.

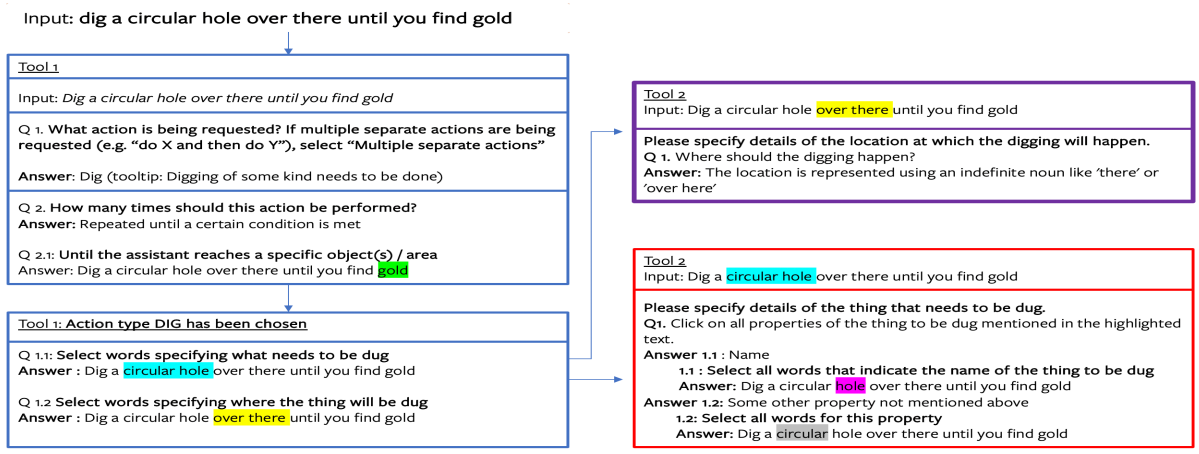

Figure 3: A representation of the annotation process using the web-based annotation tool described in Section 3.1.3. The colors of the boxes correspond to annotation tasks. The highlighting on the text in the header of the later tasks is provided by a previous annotator. We show more detailed screenshots of how the tool works in Appendix B.3

.

We emphasize that this is an *intermediate* representation. The logical forms do not come with any mechanism for generating language, and nodes do not correspond in any simple way with words. On the other hand, the logical forms do not encode all of the information necessary for execution without the use of an interpreter that can access the assistant's memory and the Minecraft world state.

Internal nodes: Internal nodes are nodes that allow recursion; although most do not require it. They can correspond to top-level actions, for example BUILD; in which case they would just be an "action" node with "action_type" build; see Figure 1. They can also correspond to arguments to top-level actions, for example a "reference_object", which specifies an object that has a spatial location. Internal nodes are not generally required to have children; it is the job of the interpreter to deal with under-specified programs like a BUILD with no arguments.

In addition to the various LOCATION, REFERENCE OBJECT, SCHEMATIC, and REPEAT nodes which can be found at various levels, another notable sub-tree is the action's STOP CONDITION, which essentially allows the agent to understand "while" loops (for example: "dig down until you hit the bedrock" or "follow me").

Leaf nodes: Eventually, arguments have to be specified in terms of values which correspond to (fixed) agent primitives. We call these nodes categorical leaves (green rectangles in Figures 1 and 2). As mentioned above, an "action" internal node has a categorical leaf child which specifies the **action type**. There are also **repeat type** nodes similarly specifying a kind of loop for example in the REPEAT sub-tree corresponding to "make three houses" the **repeat type for** specifies a "for" loop). There are also **location type** nodes specifying if a location is determined by a reference object, a set of coordinates, etc.; **relative direction** nodes that have values like "left" or "right". The complete list of categorical nodes is given in the Appendix.

However, there are limits to what we can represent with a pre-specified set of hard-coded primitives, especially if we want our agent to be able to learn new concepts or new values. Additionally, even when there is a pre-specified agent primitive, mapping some parts of the command to a specific value might be better left to an external module (e.g. mapping a number string to an integer value). For these reasons, we also have span leaves (red ovals in Figure 2). For example, in the parse for the command *"Make three oak wood houses to the left of the dark grey church."*, the SCHEMATIC (an internal node) might be specified by the command sub-string corresponding to its **name** by the span"houses" and the requested **block type** by the span "oak wood". The range of the for loop is specified by the REPEAT's **for value** ("three"), and the REFERENCE OBJECT for the location is denoted in the command by its generic **name** and specific **color** with spans "church" and "dark grey".

The root: The root of the tree has three productions: PUT_MEMORY, and GET_MEMORY, corre-

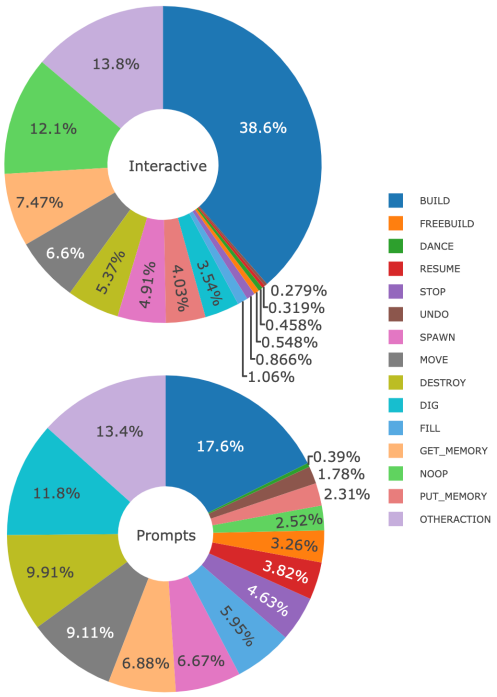

Figure 4: Frequency of each action type in the different data collection schemes described in Section 3.1.

sponding to writing to memory and reading from memory; and HUMAN_GIVE_COMMAND which also produces an ACTION_SEQUENCE, which is a special internal node whose children are ordered; multiple children correspond to an ordered sequence of commands ("build a house and then a tower"). In Figures 1 and 2 we show a schematic representation for an ACTION_SEQUENCE.

3 The CAIP Dataset

This paper introduces the CraftAssist Instruction Parsing (CAIP) dataset of English-language commands and their associated logical forms (see Appendix C for examples and a full grammar specification).

3.1 Collected Data

We collected natural language commands written by crowd-sourced workers in a variety of settings. The complete list of instructions given to crowd-workers in different settings, as well as step-by-step screen-shot of the annotation tool, are provided in the Appendix B. The basic data cleanup is described in Appendix A.

3.1.1 Image and Text Prompts

We presented crowd-sourced workers with a description of the capabilities of an assistant bot in

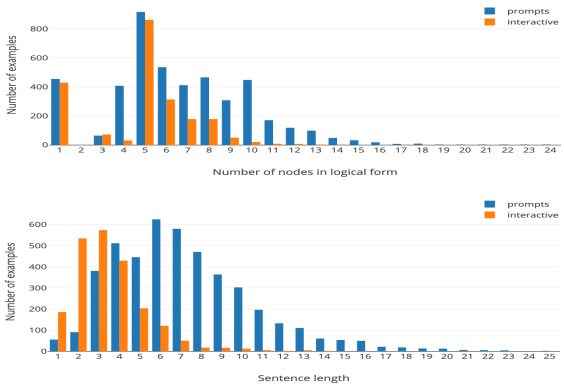

Figure 5: Histograms showing distribution over number of nodes in a logical form (top) and utterance length in words (bottom) for each data type. Prompts averages 6.74 nodes per logical form, 7.32 words per utterance, and interactive averages 4.89, 3.42 respectively

a creative virtual environment (which matches the set of allowed actions in the grammar), and (optionally) some images of a bot in a game environment. They were then asked to provide examples of commands that they might issue to an in-game assistant. We refer to these instructions as "prompts" in the rest of this paper.

3.1.2 Interactive Gameplay

We asked crowd-workers to play creative-mode Minecraft with our assistant bot, and they were instructed to use the in-game chat to direct the bot as they chose. The game sessions were capped at 10 minutes and players in this setting had no prior knowledge of the bot's capabilities or the grammar. We refer to these instructions as "Interactive" in the rest of this paper. The instructions of this setting are included in Appendix B.2.

3.1.3 Annotation Tool

Both prompts and interactive instructions come without a reference logical form and need to be annotated. To facilitate this process, we designed a multi-step web-based tool which asks users a series of multiple-choice questions to determine the semantic content of a sentence. The responses to some questions will prompt other more specific questions, in a process that mirrors the hierarchical structure of the grammar. The responses are then processed to produce the complete logical form. This allows crowd-workers to provide annotations with no knowledge of the specifics of the grammar described above. A pictorial representation of the annotation process is shown in Figure 3 and

a more detailed explanation of the process along with screen-shots of the tool is given in Appendix B.3.

We used a small set of tasks that were representative of the actual annotations to select skilled crowd-sourced workers by manually verifying the accuracy of responses on these.

Each utterance in our collection of prompts and interactive chats was shown to three different qualified annotators and we included the utterance and logical form in the dataset only if at least 2 out of 3 qualified annotators agreed on the logical form output. The total number of utterances sent to turkers was 6,775. Out of these, 6,693 had at least 2/3 agreements on the logical form and were kept. Of these, 2,872 had 3/3 agreements.

The final dataset has 4,532 annotated instructions from the prompts setting (Section 3.1.1), and 2,161 from interactive play (Section 3.1.2). The exact instructions shown to Turkers in the annotation tools are reproduced in Figures 9 and 11 in supplementary.

As in (Yih et al., 2016), we have found that careful design of the annotation tool leads to significant improvements in efficiency and accuracy. In particular, we re-affirm the conclusion from (Yih et al., 2016) that having each worker do one task (e.g. labeling a single node in the tree) makes annotation easier for workers.

3.2 Dataset Statistics

3.2.1 Action Frequencies

Since the different data collection settings described in Section 3.1 imposed different constraints and biases on the crowd-sourced workers, the distribution of actions in each subset of data is therefore different. The action frequencies of each subset are shown in Figure 4.

3.2.2 Grammar coverage

Some crowd-sourced commands describe an action that is outside the scope of the grammar. To account for this, users of the annotation tool are able to mark that a sentence is a command to perform an action that is not covered by our grammar yet. The resulting trees are labeled as OTHERACTION, and their frequency in each dataset in shown in Figure 4. Annotators still have the option to label other nodes in the tree, such as the action's LOCATION or REFERENCE OBJECT. In both the prompts and interactive data, OTHERACTION amounted to approximately 14% of the data.

3.2.3 Quantitative analysis

For each of our data types, Figure 5 show a histogram of sentence length and number of nodes. On an average interactive data has shorter sentences and smaller trees.

3.2.4 Qualitative Linguistic Style

We show the linguistic styles and choice of words of the data sources by displaying the surface forms of a set of trees. We randomly picked trees of size (number of nodes) 7 that appear in both data sources, and then for the same tree structure, we looked at the utterances corresponding to that tree. We show some representative examples in table 1. We show more examples of the data in the Appendix.

4 Related Work

There have been a number of datasets of natural language paired with logical forms to evaluate semantic parsing approaches, e.g. (Price, 1990; Tang and Mooney, 2001; Cai and Yates, 2013; Wang et al., 2015; Zhong et al., 2017). The dataset presented in this work is an order of magnitude larger than those in (Price, 1990; Tang and Mooney, 2001; Cai and Yates, 2013) and is similar in scale to the datasets in (Wang et al., 2015), but smaller than (Zhong et al., 2017).

In addition to mapping natural language to logical forms, our dataset connects both of these to a dynamic environment. In (Lauria et al., 2001; Bos and Oka, 2007; Tellex et al., 2011; Matuszek et al., 2013; Thomason et al., 2019) semantic parsing has been used for interpreting natural language commands for robots. In our paper, the "robot" is embodied in the Minecraft game instead of in the physical world. In (Boye et al., 2006) semantic parsing has been used for spoken dialogue with an embodied character in a 3-D world with pattern matching and rewriting phases. In our work, the user along with the assistant is embodied in game and instructs using language. We go from language to logical forms end-to-end with no pattern match necessary. Semantic parsing in a voxel-world recalls (Wang et al., 2017), where the authors describe a method for building up a programming language from a small core via interactions with players.

We demonstrate the results of several neural parsing models on our dataset. In particular, we show the results of a re-implementation of (Dong

Prompts	bot move to where the tree is	dig a large size hole to put these waste particles into the hole	please build a sphere on that location	hey bot can you dig a 5 by 5 hole for me
Interactive	find tree	dig large hole	build a sphere over here	dig a 5 x 5 hole

Table 1: Choice of words across different data sources for the same logical form (per column).

and Lapata, 2016) adapted to our grammar, and a straightforward fine-tuned BERT model (Devlin et al., 2018). There have been several other papers proposing neural architectures for semantic parsing, for example (Jia and Liang, 2016; Zhong et al., 2017; Wang et al., 2018; Hwang et al., 2019); in particular (Hwang et al., 2019) uses a BERT based model. In those papers, as in this one, the models are trained with full supervision of the mapping from natural language to logical forms, without considering the results of executing the logical form (in this case, the effect on the environment of executing the actions denoted by the logical form). There has been progress towards "weakly supervised" semantic parsing (Artzi and Zettlemoyer, 2013; Liang et al., 2016; Guu et al., 2017) where the logical forms are hidden variables, and the only supervision given is the result of executing the logical form. There are now approaches that have shown promise without even passing through (discrete) logical forms at all (Riedel et al., 2016; Neelakantan et al., 2016). We hope that the dataset introduced here, which has supervision at the level of the logical forms, but whose underlying grammar and environment can be used to generate essentially infinite weakly supervised or execution rewards, will also be useful for studying these models.

Minecraft, especially via the MALMO project (Johnson et al., 2016) has been used as a base environment for several machine learning papers. It is often used as a testbed for reinforcement learning (RL) (Shu et al., 2017; Udagawa et al., 2016; Alaniz, 2018; Oh et al., 2016; Tessler et al., 2017). In these works, the agent is trained to complete tasks by issuing low level actions (as opposed to our higher level primitives) and receiving a reward on success. Others have collected large-scale datasets for RL and imitation learning (Guss et al., 2019a,b). Some of these works (e.g. (Oh et al., 2017)) do consider simplified, templated language as a method for composably specifying tasks, but training an RL agent to execute the scripted primitives in our grammar is already nontrivial, and so the task space and language in those works is more constrained than what we use here. Nevertheless, our work may be useful to researchers interested in RL (or imitation): using our grammar and executing in game can supply (hard) tasks and descriptions, and demonstrations. Another set of works (Kitaev and Klein, 2017; Yi et al., 2018) have used Minecraft for visual question answering with logical forms. Our work extends these to interactions with the environment. Finally, (Allison et al., 2018) is a more focused study on how a human might interact with a Minecraft agent; our collection of free generations (see 3.1.1) includes annotated examples from similar studies of players interacting with a player pretending to be a bot.

5 Baseline Models

In order to assess the challenges of the dataset, we implement two models which learn to read a sentence and output a logical form by formulating the problem as a sequence-to-tree and a sequence-to-sequence prediction task respectively.

5.1 Sequence to Tree Model

Our first model adapts the Seq2Tree approach of (Dong and Lapata, 2016) to our grammar. In short, a bidirectional RNN encodes the input sentence into a sequence of vectors, and a decoder recursively predicts the tree representation of the logical form, starting at the root and predicting all of the children of each node based on its parent and left siblings and input representation.

Sentence Encoder and Attention: We use a bidirectional GRU encoder (Cho et al., 2014) which encodes a sentence of length T $\mathbf{s} = (w_1, \dots\ w_T)$ into a sequence of T dimension d vectors:

$$f_{GRU}(\mathbf{s}) = (\mathbf{h}_1, \dots, \mathbf{h}_T) \in \mathbb{R}^{d \times T}$$

Tree Decoder: The decoder starts at the root, computes its node representation and predicts the state of its children, then recursively computes the representations of the predicted descendants. Similarly to Seq2Tree, a node representation $\mathbf{r}_n$ is computed based on its ancestors and left siblings. We also found it useful to condition each of the node

representation on the encoder output explicitly for each node. Thus, we compute the representation $\mathbf{r}_{n_t}$ and recurrent hidden state $\mathbf{g}_{n_t}$ for node n_t as:

$$\mathbf{r}_{n_t} = \text{attn}(\mathbf{v}_{n_t} + \mathbf{g}_{n_{t-1}}, (\mathbf{h}_1, \ldots, \mathbf{h}_T);\ \mathbf{M}^\sigma) \quad (1)$$

$$\mathbf{g}_{n_t} = f_{rec}(\mathbf{g}_{n_{t-1}}, (\mathbf{v'}_{n_t} + \mathbf{r}_{n_t})) \quad (2)$$

Where attn is multi-head attention, $\mathbf{M}^\sigma \in \mathbb{R}^{d\times d\times K}$ is a tree-wise parameter, f_{rec} is the GRU recurrence function, and $\mathbf{v'}_{n_t}$ is a node parameter (one per category for categorical nodes), and n_{t-1} denotes either the last predicted left sibling if there is one or the parent node otherwise.

Prediction Heads: Finally, the decoder uses the computed node representations to predict the state of each of the internal, categorical, and span nodes in the grammar. We denote each of these sets by $\mathcal{I}$, $\mathcal{C}$ and $\mathcal{S}$ respectively, and the full set of nodes as $\mathcal{N} = \mathcal{I} \cup \mathcal{C} \cup \mathcal{S}$.

First, each node in $\mathcal{N}$ is either active or inactive in a specific logical form. We denote the state of a node n by $a_n \in \{0, 1\}$. All the descendants of an inactive internal node $n \in \mathcal{I}$ are considered to be inactive. Additionally, each categorical node $n \in \mathcal{C}$ has a set of possible values C^n; its value in a specific logical form is denoted by the category label $c_n \in \{1, \ldots, |C^n|\}$. Finally, active span nodes $n \in \mathcal{S}$ for a sentence of length T have a start and end index $(s_n, e_n) \in \{1, \ldots, T\}^2$. We compute, the representations $\mathbf{r}_n$ of the nodes as outlined above, then obtain the probabilities of each of the labels by:

$$\forall n \in \mathcal{N}, \quad p(a_n) = \sigma(\langle \mathbf{r}_n, \mathbf{p}_n \rangle) \quad (3)$$

$$\forall n \in \mathcal{C}, \quad p(c_n) = \text{softmax}(M^c_n \mathbf{r}_n) \quad (4)$$

$$\forall n \in \mathcal{S}, \quad p(s_n) = \text{softmax}(\mathbf{r}_n^\mathsf{T} M^s_n(\mathbf{h}_1, \ldots, \mathbf{h}_T))$$
$$p(e_n) = \text{softmax}(\mathbf{r}_n^\mathsf{T} M^e_n(\mathbf{h}_1, \ldots, \mathbf{h}_T)) \quad (5)$$

where the following are model parameters:

$$\forall n \in \mathcal{N}, \quad \mathbf{p}_n \in \mathbb{R}^d$$
$$\forall n \in \mathcal{C}, \quad M^c_n \in \mathbb{R}^{d\times d}$$
$$\forall n \in \mathcal{S}, \quad (M^s_n, M^e_n)_n \in \mathbb{R}^{d\times d\times 2}$$

Let us note the parent of a node n as $\pi(n)$. Given Equations 3 to 5, the log-likelihood of a tree with states $(\mathbf{a}, \mathbf{c}, \mathbf{s}, \mathbf{e})$ given a sentence $\mathbf{s}$ is then:

$$\mathcal{L} = \sum_{n\in\mathcal{N}} a_{\pi(n)} \log(p(a_n)) + \sum_{n\in\mathcal{C}} a_n \log(p(c_n)) + \sum_{n\in\mathcal{S}} a_n \Big(\log(p(s_n)) + \log(p(e_n)) \Big) \quad (6)$$

Overall, our implementation differs from the original Seq2Tree in three ways, which we found lead to better performance in our setting. First, we replace single-head with multi-head attention. Secondly, the cross-attention between the decoder and attention is conditioned on both the node embedding and previous recurrent state. Finally, we replace the categorical prediction of the next node by a binary prediction problem: since we know which nodes are eligible as the children of a specific node (see Figures 1 and 2), we find that this enforces a stronger prior. We refer to this modified implementation as SentenceRec.

5.2 Sequence to Sequence Model

Our second approach treats the problem of predicting the logical form as a general sequence-to-sequence (Seq2Seq) task; such approaches have been used in semantic parsing in e.g. (Jia and Liang, 2016; Wang et al., 2018). We take the approach of (Jia and Liang, 2016) and linearize the output trees: the target sequence corresponds to a Depth First Search walk through the tree representation of the logical form. More specifically the model needs to predict, in DFS order, a sequence of tokens corresponding to opening and closing internal nodes, categorical leaves and their value, and span leaves with start and end sequences. In practice, we let the model predict span nodes in two steps: first predict the presence of the node, then predict the span value, using the same prediction heads as for the SentenceRec model (see Equation 5 above). With this formalism, the logical form for e.g. *"build a large blue dome on top of the walls"* will be:

```
(ACTION_TYPE:BUILD, OPEN:SCHEMATIC,
   HAS_SIZE, SIZE_SPAN-(2,2),
   HAS_COLOR, COLOR_SPAN-(3,3),
   HAS_NAME, NAME_SPAN-(4,4),
 CLOSE:SCHEMATIC, OPEN:LOCATION,
   LOC_TYPE:REF_OBJECT, REL_DIR:UP,
   OPEN:REF_OBJECT,
     HAS_NAME, NAME_SPAN-(9,9),
   CLOSE:REF_OBJECT,
 CLOSE:LOCATION)
```

We train a BERT encoder-decoder architecture on this sequence transduction task, where the training loss is a convex combination of the output sequence log-likelihood and the span cross-entropy loss.

Pre-trained Sentence Encoder: Finally, recent work has shown that using sentence encoder that has been pre-trained on large-scale language modeling tasks can lead to substantial performance

	Acc. (std)	Inter.	Prompts
SentRec	50.08 (2.97)	64.17	42.49
DistBERT+SentRec	59.58 (3.49)	76.0	50.74
DistBERT+Seq2Seq	**60.74** (3.58)	76.06	52.49

Table 2: Average accuracy over a test set of 650 Prompts + 350 Interactive.

	N=2	N=5	N=15
Joint	67.7	72.76	75.7
Interactive	83.83	88.34	90.63
Prompts	59.02	64.37	67.66

Table 3: Recall at N for the Seq2Seq model beam search.

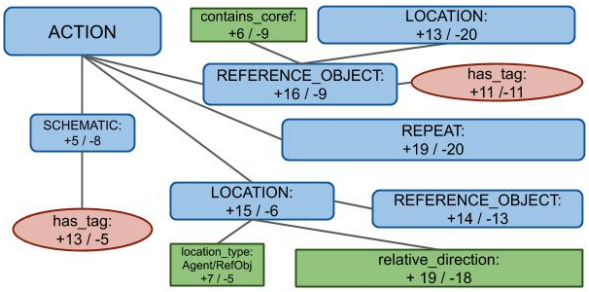

Figure 6: We show nodes in the grammar which are most often wrongly predicted, with false positive (+) and false negative counts (-).

improvements (Song et al., 2019).We use the pre-trained DistilBERT model of (Sanh et al., 2019) as the encoder of our sequence-to-sequence model, and also propose a version of the SentenceRec which uses it to replace the bidirectional RNN.

6 Experiments

In this Section, we evaluate the performance of our baseline models on the proposed dataset.

Training Data: The CAIP datasets consists in a total of 6693 annotated instruction-parse pairs. In order for our models to make the most of this data while keeping the evaluation statistically significant, we create 5 different train/test splits of the data and report the average performance of models trained and evaluated on each of them. In each case, we hold out 650 examples from Prompts and 350 from Interactive for testing, and use the remaining 5693 as the training set.

Modeling Choices: For the end-to-end trained SentenceRec model, we use a 2-layer GRU sentence encoder and all hidden layers have dimension $d = 256$. We use pre-trained word embeddings computed with FastText with subword information (Bojanowski et al., 2017). The decoder uses a GRU recurrent cell and 4-headed attention. The Seq2Seq model uses a variant of the *bert-base-uncased* provided in the Transformer library [4] with 6 encoding and decoding layers. For the Seq2Seq model and the SentenceRec with pre-trained encoder, we use the *distilbert-base-uncased* encoder from the same library. The Seq2Seq model uses beam search decoding with 15 beams. All models are trained with the Adam optimizer with quadratic learning rate decay. We provide our model and training code along with the dataset for reproducibility purposes.

Overview of Results: Table 2 provides the average accuracy (computed as the proportion of logical forms that are entirely accurately predicted) and standard deviation across all five splits, as well as the contributions of the Interactive and Prompts data. The first observation is that using a pre-trained encoder leads to a significant improvement, with a 10 point boost in accuracy. On the other hand, while the Seq2Seq model is more general and makes less use of our prior knowledge of the structure of logical forms, it does marginally better than the recursive prediction model (although within one standard deviation).

Secondly, although the models are trained on more data provided from the Prompts setting than from Interactive play, they all do better on the latter. This is consistent with previous observations on the dataset statistics in Section 3.2.3 which find that players tend to give shorter instructions with simpler execution. Finally, we note that one of the advantages of having the parser be part of an interactive agent is that it can ask the player for clarification and adapt its behavior when it is made aware of a mistake (Yao et al., 2019). In that spirit, Table 3 provides Recall at N numbers, which represent how often the true parse is within the N first elements of the beam after beam search. Recall at 2 does provide a consistent boost over the accuracy of a single prediction, but even the full size 15 beam does not always contain the right logical form.

Error Analysis: We further investigate the errors of the Seq2seq models on one of the data splits. We find that the model still struggles with span predictions: out of 363 errors, 125 only make mistakes on spans (and 199 get the tree structure right

[4] https://github.com/huggingface/transformers

but make mistakes on leaves). Figure 6 shows the nodes which are most commonly mistaken, with the number of false positive and false negatives out of these 363 mistakes. Unsurprisingly, the most commonly confused span leaf is "has_tag", which we use as a miscellaneous marker. Aside from that "has_tag" however, the span mistakes are evenly spread over all other leaves. The next most common source of mistakes comes from the model struggling between identifying whether a provided location corresponds to the target of the action or to the reference object, and to identify instructions which imply a repetition. The former indicates a lack of compositionality in the input representation: the model correctly identifies that a location is mentioned, but fails to identify its context. Repeat conditions on the other hand challenge the model due to the wide variety of possible stop condition, a problem we suggest future work pay special attention to.

7 Conclusion

In this work, we have described a grammar over a mid-level interface for a Minecraft assistant. We then discussed the creation of a dataset of natural language utterances with associated logical forms over this grammar that can be executed in-game. Finally, we showed the results of using this new dataset to train several neural models for parsing natural language instructions. Consistently with recent works, we find that BERT pre-trained models do better than models trained from scratch, but there is much space for improvement. We believe this data will be useful to researchers studying semantic parsing, especially interactive semantic parsing, human-robot interaction, and even imitation and reinforcement learning.

References

Stephan Alaniz. 2018. Deep reinforcement learning with model learning and monte carlo tree search in minecraft. *arXiv preprint arXiv:1803.08456*.

Fraser Allison, Ewa Luger, and Katja Hofmann. 2018. How players speak to an intelligent game character using natural language messages. *Transactions of the Digital Games Research Association*, 4(2).

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.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *TACL*, 5:135–146.

Johan Bos and Tetsushi Oka. 2007. A spoken language interface with a mobile robot. *Artificial Life and Robotics*, 11(1):42–47.

Johan Boye, Joakim Gustafson, and Mats Wirén. 2006. Robust spoken language understanding in a computer game. *Speech Commun.*, 48:335–353.

Qingqing Cai and Alexander Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 423–433.

Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S Lam. 2017. Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant. In *Proceedings of the 26th International Conference on World Wide Web*, pages 341–350.

Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S Lam. 2019. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 394–410.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734.

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

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280*.

Jonathan Gray, Kavya Srinet, Yacine Jernite, Haonan Yu, Zhuoyuan Chen, Demi Guo, Siddharth Goyal, C Lawrence Zitnick, and Arthur Szlam. 2019. Craftassist: A framework for dialogue-enabled interactive agents. *arXiv preprint arXiv:1907.08584*.

William H Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. 2019a. The minerl competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*.

William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. 2019b. Minerl: a large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *arXiv preprint arXiv:1704.07926*.

Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *arXiv preprint arXiv:1902.01069*.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. 2016. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247.

Nikita Kitaev and Dan Klein. 2017. Where is misty? interpreting spatial descriptors by modeling regions in space. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 157–166.

Thomas Kollar, Danielle Berry, Lauren Stuart, Karolina Owczarzak, Tagyoung Chung, Lambert Mathias, Michael Kayser, Bradford Snow, and Spyros Matsoukas. 2018. The alexa meaning representation language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, volume 3, pages 177–184.

Thomas Kollar, Jayant Krishnamurthy, and Grant P Strimel. 2013. Toward interactive grounded language acqusition. In *Robotics: Science and systems*, volume 1, pages 721–732.

Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, Johan Bos, and A Klein. 2001. Training personal robots using natural language instruction. *IEEE Intelligent systems*, 16(5):38–45.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. 2016. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*.

Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. 2013. Learning to parse natural language commands to a robot control system. In *Experimental Robotics*, pages 403–415. Springer.

Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. 2016. Learning a natural language interface with neural programmer. *arXiv preprint arXiv:1611.08945*.

Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. 2016. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. 2017. Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*.

Patti J Price. 1990. Evaluation of spoken language systems: The atis domain. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990*.

Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. 2016. Programming with a differentiable forth interpreter. *CoRR, abs/1605.06640*.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108.

Tianmin Shu, Caiming Xiong, and Richard Socher. 2017. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *arXiv preprint arXiv:1712.07294*.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: masked sequence to sequence pre-training for language generation. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 5926–5936.

Lappoon R Tang and Raymond J Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *European Conference on Machine Learning*, pages 466–477. Springer.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth Teller, and Nicholas Roy. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J Mankowitz, and Shie Mannor. 2017. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, volume 3, page 6.

Jesse Thomason, Aishwarya Padmakumar, Jivko Sinapov, Nick Walker, Yuqian Jiang, Harel Yedidsion, Justin Hart, Peter Stone, and Raymond J Mooney. 2019. Improving grounded natural language understanding through human-robot dialog. *arXiv preprint arXiv:1903.00122*.

Jesse Thomason, Shiqi Zhang, Raymond J Mooney, and Peter Stone. 2015. Learning to interpret natural language commands through human-robot dialog. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Hiroto Udagawa, Tarun Narasimhan, and Shim-Young Lee. 2016. Fighting zombies in minecraft with deep reinforcement learning. Technical report, Technical report, Stanford University.

Sida I Wang, Samuel Ginn, Percy Liang, and Christoper D Manning. 2017. Naturalizing a programming language via interactive learning. pages 929–938.

Sida I Wang, Percy Liang, and Christopher D Manning. 2016. Learning language games through interaction. *arXiv preprint arXiv:1606.02447*.

Wenlu Wang, Yingtao Tian, Hongyu Xiong, Haixun Wang, and Wei-Shinn Ku. 2018. A transfer-learnable natural language interface for databases. *arXiv preprint arXiv:1809.02649*.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1332–1342.

Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based interactive semantic parsing: A unified framework and A text-to-sql case study. *CoRR*, abs/1910.05389.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. 2018. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pages 1039–1050.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Basic Data Cleanup

We threw away all duplicate commands in the dataset and only got annotations for unique commands from each data source.

We performed post-processing on the text by first inserting spaces between any special character (brackets, “,”, “x”) followed by alphanumeric character. For example “make a 5x5 hole” was post-processed to “make a 5 x 5 hole” and “go to (1,2,3)” to “go to (1 , 2 , 3)”. We then used the tokenizer from spaCy [5] to tokenize every word in the sentence.

When constructing logical forms: we threw away any keys with values : ‘None’ , ‘Other’ or ‘Not Specified’ . Our tool allows workers to select these options when annotating. We skipped stopwords and articles like ‘a’ , ‘an’ etc when constructing spans of children. We reordered the indices of words in spans to always be from left to right (regardless of which order the words were selected in the sentence when annotating).

For commands annotated as “composite” (meaning a command that requires multiple actions), we set up another tool where we asked crowd-sourced workers to split the composite command into individual commands. Each of these commands were then sent to our web-based tool described in 3.1.3 and the results were combined together under the key: “action_sequence” by preserving the order. So in the sentence: “jump twice and then come to me”, we first have the sentence split into commands: “jump twice” and “come to me” and then combine their logical forms together under “action_sequence” so we first have the “Dance” action followed by “Move” action. This tool is described in Section B.4.

B Crowd-sourced task and tools instructions

This section covers details of each crowd sourced task we’ve described in the paper along with screenshots of the web-based annotation tool described in 3.1.

B.1 Image and Text Prompts

In this task we showed a screenshot of the bot and environment to the crowd-sourced workers and asked them to give us free-form commands for the assistant. The instructions shown to workers are shown in 7.

[5] https://spacy.io/

Figure 7: The task instructions shown to crowd-sourced workers for the Image and text prompts task

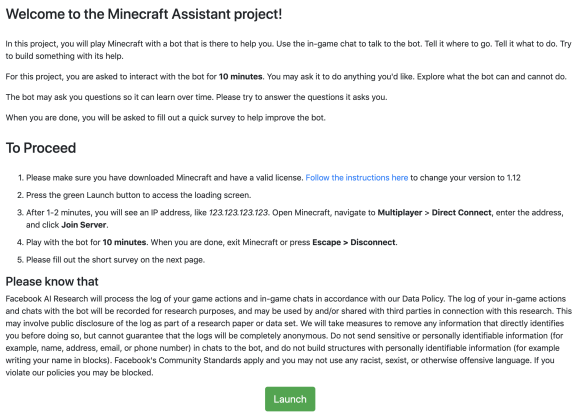

Figure 8: The task instructions shown to crowd-sourced workers for the interactive game play

B.2 Interactive Gameplay

In this task we had crowd-sourced workers play with our bot and interact with it using in-game chat. The instructions shown to workers are shown in 8.

B.3 Annotation tool

The web based annotation tool has two subparts: Tool a and Tool b.

B.3.1 Tool a

This tool is the first tool in the process of annotation and asks crowd-sourced workers to help determine the intent (dialogue_type or action_type) of the sentence and highlight other pieces of the text based on the choices they made for the intent. (For example: if the intent was “Build” they are asked to select words for the thing to be built and the location respectively.) We also provided helpful tooltips with examples at every step of the process.

The instructions shown to workers for Tool a are shown in figure 9 and step by step annotation process is shown in figure 10

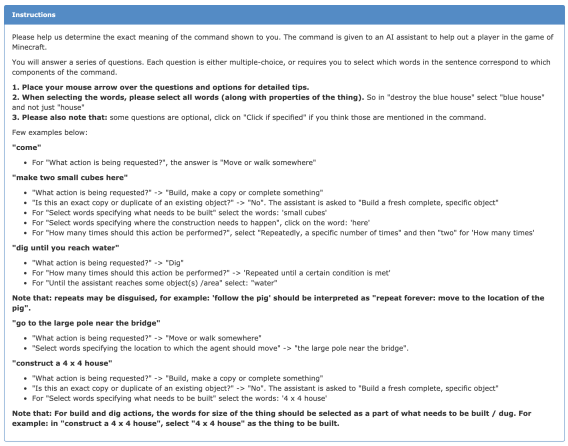

Figure 9: The task instructions shown to crowdsourced workers for the annotation Tool a

B.3.2 Tool b

After we determine the intent from Tool a and get highlighted span of words for respective children of the intent, we use this tool. This is the second tool in the annotation process and asks crowdsourced workers to help determine the fin-grained properties of specific entities of the action or dialogue. Note that we already got the words representing these, highlighted in B.3.1. For example : the words " big bright house" are highlighted in the sentence "destroy the big bright house by the tree " as an outcome of Tool a. The questionnaire changes dynamically based on the choices the workers make at every step of the tool. We provided helpful tooltips with examples at every step of the annotation process. Using the output of Tool a and Tool b, we can successfully construct the entire logical form for a given sentence.

The instructions shown to workers for Tool b are shown in Figure 11 and step by step annotation process for annotating properties of "location" in a "Move" action is shown in Figure 12 and annotating "reference_object" in "Destroy" action is shown in Figure 13

B.4 Tool for composite commands

This tool is meant for "composite" commands (commands that include multiple actions) and asks the users to split a command into multiple individual commands. The instruction for this are shown in figure 14. Once we get the split, we send out each command to annotation tool described in Section B.3

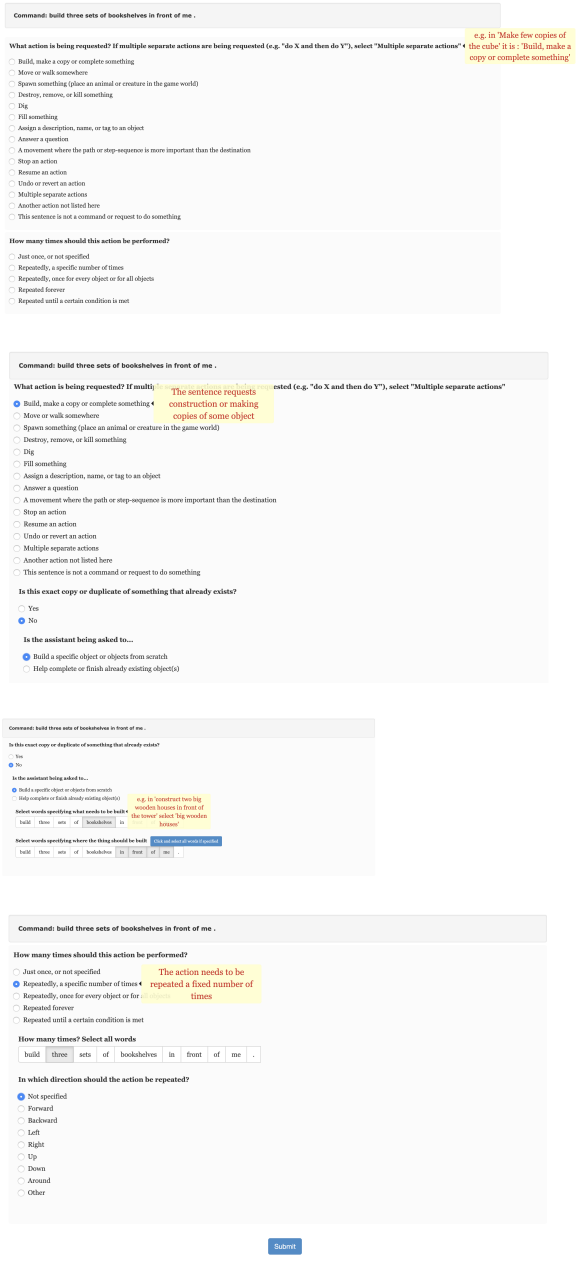

Figure 10: The step by step screenshot of annotations process for the command: "build three sets of bookshelves in front of me ." in Tool a

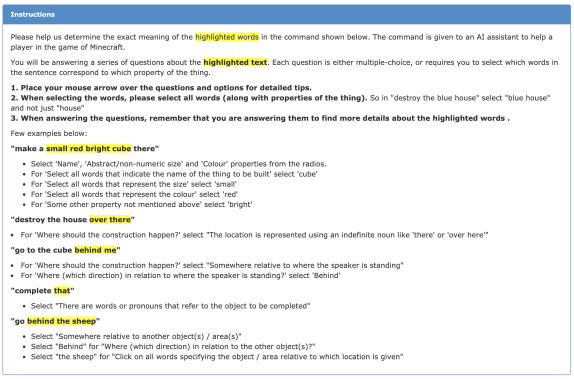

Figure 11: The task instructions shown to crowd-sourced workers for the annotation Tool b

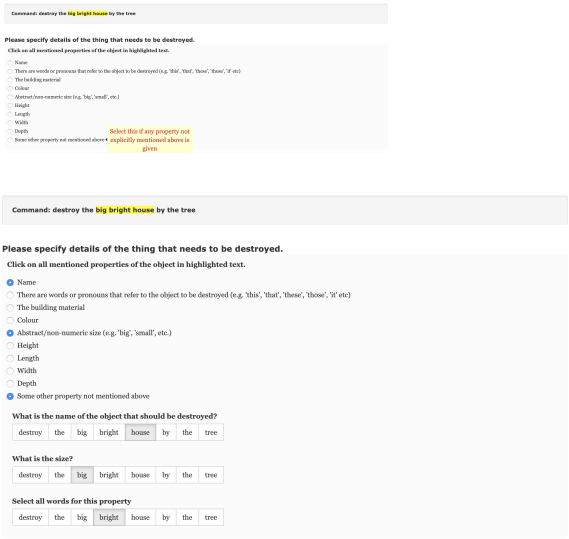

Figure 13: The step by step screenshot of annotating properties of highlighted words for"reference_object" in a "Destroy" action.

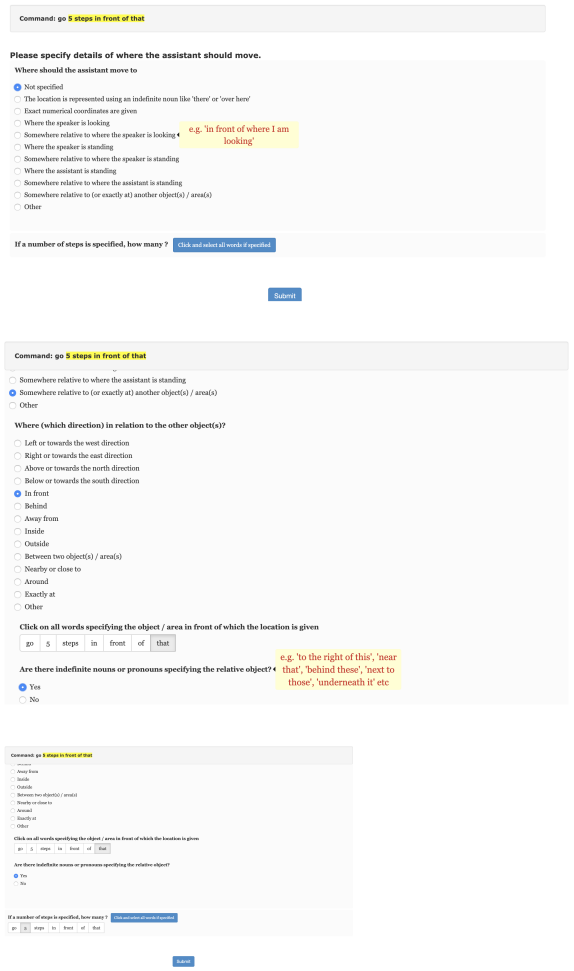

Figure 12: The step by step screenshot of annotating properties of highlighted words for"location" in a "Move" action.

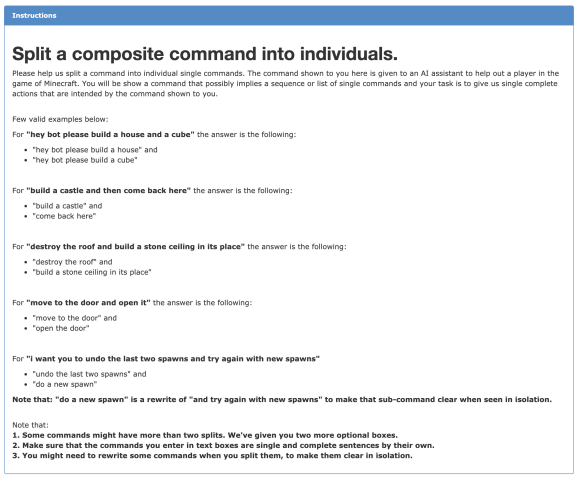

Figure 14: The task instructions shown to crowd-sourced workers for splitting composite commands

C Action Tree structure

This section describes the details of logical form of each action. We support three dialogue types: HUMAN_GIVE_COMMAND, GET_MEMORY and PUT_MEMORY. The logical form for actions has been pictorially represented in Figures: 1 and 2

We support the following actions in our dataset : Build, Copy, Dance, Spawn, Resume, Fill, Destroy, Move, Undo, Stop, Dig and FreeBuild. A lot of the actions use "location" and "reference_object" as children in their logical forms. To make the logical forms more presentable, we have shown the detailed representation of a "reference_object" (reused in action trees using the variable: "REF_OBJECT") in Figure 15 and the representation of "location" (reused in action trees using the variable: "LOCATION") in figure 16. The representations of actions refer to these variable names in their trees.

```
REF_OBJECT :
The recursion depth of REF_OBJECT in LOCATION
was never greater than 1 in the data. So a REF_OBJECT
can have a LOCATION that has a REF_OBJECT that has a
LOCATION (and the final location will be one of :
COORDINATES / AGENT_POS / SPEAKER_POS / SPEAKER_LOOK).

"reference_object" : {
 "repeat" : {
  "repeat_key" : 'FOR' / 'ALL',
  "repeat_count" : span,
   "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
      'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
  "has_name" : span,
  "has_colour" : span,
  "has_size" : span,
  "has_tag": span,
  "has_length": span,
  "has_width": span,
  "has_height": span,
  "contains_coreference" : "yes",
  LOCATION  }
```

Figure 15: Logical form of a reference_object child

```
LOCATION:

"location" : {
 "location_type" : COORDINATES / REFERENCE_OBJECT /
    AGENT_POS / SPEAKER_POS / SPEAKER_LOOK
 "steps" : span,
 "contains_coreference" : "yes",
 "relative_direction" : 'LEFT' / 'RIGHT' / 'UP'/
      'DOWN' / 'FRONT' / 'BACK' / 'AWAY' / 'INSIDE'
      / 'NEAR' / 'OUTSIDE' / 'BETWEEN',
 "coordinates" : span, (present if "location_type" is
   'COORDINATES),
  REF_OBJECT (present if "location_type" is
   'REFERENCE_OBJECT')
 }
```

Figure 16: Logical form of a location child

The detailed action tree for each action and dialogue type has been presented in the following subsections. Figure 17 shows an example for a BUILD action.

```
  0     1    2   3     4    5   6
"Make three oak wood houses to the
 7   8   9   10   11    12
left of the dark grey church."


{"dialogue_type" : "HUMAN_GIVE_COMMAND",
 "action_sequence" : [
   {
    "action_type" : "BUILD",
    "schematic": {
      "has_block_type": [0, [2, 3]],
      "has_name": [0, [4, 4]],
      "repeat": {
        "repeat_key": "FOR",
        "repeat_count": [1, 1]
      }},
     "location": {
       "relative_direction": "LEFT",
       "location_type": "REFERENCE_OBJECT",
       "reference_object": {
         "has_colour_": [0, [10, 11]],
          "has_name_": [0, [12, 12]] }
}}]}
```

Figure 17: An example logical form. The spans are indexed as : [sentence_number, [starting_word_index, ending_word_index]]. sentence_number is 0 for the most recent sentence spoken in a dialogue and is 0 in our dataset since we support one-turn dialogues as of now.

C.1 Build Action

This is the action to Build a schematic at an optional location. The Build logical form is shown in 18 .

C.2 Copy Action

This is the action to copy a block object to an optional location. The copy action is represented as a "Build" with an optional "reference object" . The logical form is shown in 19.

C.3 Spawn Action

This action indicates that the specified object should be spawned in the environment. The logical form is shown in: 20

C.4 Fill Action

This action states that a hole / negative shape at an optional location needs to be filled up. The logical form is explained in : 21

C.5 Destroy Action

This action indicates the intent to destroy a block object at an optional location. The logical form is shown in: 22

Destroy action can have one of the following as the child:

- reference object

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'BUILD',
    LOCATION,
    "schematic" : {
      "repeat" : {
       "repeat_key" : 'FOR' / 'ALL',
       "repeat_count" : span,
       "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
          'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
      "has_name" : span,
      "has_block_type" : span,
      "has_size" : span,
      "has_orientation" : span,
      "has_thickness" : span,
      "has_colour" : span,
      "has_length": span,
      "has_height" : span,
      "has_radius" : span,
      "has_slope" : span,
      "has_width": span,
      "has_base" : span,
      "has_distance" : span,
     },
    "repeat" : {
      "repeat_key" : 'FOR' / 'ALL',
       "repeat_count" : span,
       "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
          'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
     } ] }
```

Figure 18: Details of logical form for Build

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'BUILD',
    LOCATION,
    REF_OBJ,
    "repeat" : {
      "repeat_key" : 'FOR' / 'ALL',
       "repeat_count" : span,
       "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
          'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
     } ] }
```

Figure 19: Details of logical form for Copy

- nothing

C.6 Move Action

This action states that the agent should move to the specified location, the corresponding logical form is in: 23

Move action can have one of the following as its child:

- location
- stop condition (stop moving when a condition is met)
- location and stop condition
- neither

C.7 Dig Action

This action represents the intent to dig a hole / negative shape of optional dimensions at an optional location. The logical form is in 24

C.8 Dance Action

This action represents that the agent performs a movement of a certain kind. Note that this action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'SPAWN',
    LOCATION,
    REF_OBJ } ] }
```

Figure 20: Details of logical form for Spawn action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'FILL',
    "has_block_type" : span,
    REF_OBJ } ] }
```

Figure 21: Details of logical form for Fill

is different than a Move action in that the path or step-sequence here is more important than the destination. The logical form is shown in 25

C.9 FreeBuild Action

This action represents that the agent should complete an already existing half-finished block object, using its mental model. The logical form is explained in: 26

FreeBuild action can have one of the following as its child:

- reference object only
- reference object and location

C.10 Undo Action

This action states the intent to revert the specified action, if any. The logical form is in 27. Undo action can have on of the following as its child:

- target_action_type
- nothing (meaning : undo the last action)

C.11 Stop Action

This action indicates stop and the logical form is shown in 28

C.12 Resume Action

This action indicates that the previous action should be resumed, the logical form is shown in: 29

C.13 Get Memory Dialogue type

This dialogue type represents the agent answering a question about the environment. This is similar to the setup in Visual Question Answering. The logical form is represented in: 30

Get Memory dialogue has the following as its children: filters, answer type and tag name. This dialogue type represents the type of expected answer : counting, querying a specific attribute or querying everything (”what is the size of X” vs ”what is X”)

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'DESTROY',
    REF_OBJ } ] }
```

Figure 22: Details of logical form Destroy

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'MOVE',
    LOCATION,
    "stop_condition" : {
     "condition_type": 'ADJACENT_TO_BLOCK_TYPE' /
           'NEVER',
     "block_type": span,
     "condition_span" :  span },
    "repeat" : {
      "repeat_key" : 'FOR' / 'ALL',
       "repeat_count" : span,
       "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
          'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
     } ] }
```

Figure 23: Details of logical form for Move action

C.14 Put Memory Dialogue

This dialogue type represents that a reference object should be tagged with the given tag and the logical form is shown in: 31

C.15 Noop Dialogue

This dialogue type indicates no operation should be performed, the logical form is shown in : 32

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'DIG',
    LOCATION,
    "schematic" : {
       "repeat" : {
         "repeat_key" : 'FOR' / 'ALL',
         "repeat_count" : span,
         "repeat_dir" : 'LEFT' / 'RIGHT' / 'UP'/
          'DOWN' / 'FRONT' / 'BACK' / 'AROUND'}
      "has_size" : span,
      "has_length": span,
      "has_depth" :  span,
      "has_width" :  span
       },
    "stop_condition" : {
      "condition_type" : 'ADJACENT_TO_BLOCK_TYPE' /
          'NEVER',
      "block_type": span }
     } ] }
```

Figure 24: Details of logical form for Dig action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'DANCE',
    LOCATION,
    "stop_condition" : {
      "condition_type" : 'NEVER'}
    "repeat: {
      "repeat_key" : FOR,
      "repeat_count" :  span }
     } ] }
```

Figure 25: Details of logical form for Dance action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'FREEBUILD',
    REF_OBJECT,
    LOCATION }  ]  }
```

Figure 26: Details of logical form for Freebuild action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'UNDO',
    "target_action_type" : span } ] }
```

Figure 27: Details of logical form for Undo action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'STOP',
    "target_action_type" : span } ] }
```

Figure 28: Details of logical form for Stop action

```
{ "dialogue_type" :  'HUMAN_GIVE_COMMAND',
 "action_sequence" : [
   {"action_type" :  'RESUME',
    "target_action_type" : span } ] }
```

Figure 29: Details of logical form for Resume action

```
{ "dialogue_type": "GET_MEMORY",
  "filters": {
    "temporal": CURRENT,
    "type": "ACTION" / "AGENT" / "REFERENCE_OBJECT",
    "action_type": BUILD / DESTROY / DIG / FILL /
        SPAWN / MOVE
    "reference_object" : {
      LOCATION,
      "has_size" : span,
      "has_colour" : span,
      "has_name" : span,
      "coref_resolve": span,
      },
    },
  "answer_type": "TAG" / "EXISTS" ,
  "tag_name" : 'has_name' / 'has_size' /'has_colour' /
       'action_name' / 'action_reference_object_name' /
       'move_target' / 'location' ,
  "replace": true
}
```

Figure 30: Details of logical form for Get Memory Dialogue

```
{ "dialogue_type": "PUT_MEMORY",
  "filters": { REF_OBJECT },
  "upsert" : {
      "memory_data": {
        "memory_type": "REWARD" / "TRIPLE",
        "reward_value": "POSITIVE" / "NEGATIVE",
        "has_tag" : span,
        "has_colour": span,
        "has_size": span
      } } }
```

Figure 31: Details of logical form for Put Memory Dialogue

```
{ "dialogue_type": "NOOP" }
```

Figure 32: Details of logical form for Noop Dialogue