Grammatically Constrained Language Grounding

Cobi Finkelstein [cefinkelstei] **Ray Huang** zixuanhuang **Chunghin Lee** chunghinlee **Harsha Malireddy** smalireddy **Jiaming Yuan** jiamingyuan]

1 Problem statement

To enable humans to talk to robots, natural language (NL) commands must be grounded into a symbolic goal. However, NL is often ambiguous and commands are often nuanced, making language grounding errors and the robot mistakes inevitable. The current state of the art grounds these commands into linear temporal logic (LTL). The commands are grounded via a neural network sequence-to-sequence model with attention that is trained to handle arbitrary NL commands using data from a crowd-sourced corpus.

The latest language grounding model (Oh et al., 2019) only achieves about 80% accuracy on the unseen commands for their data-set– with no error recovery technique for the remaining 20%. Danas et al. (2019) introduce a dialogue based recovery technique, that weakens the correctness criteria on the language grounding model to produce the correct grounding in the top-k searched beams. However, this technique requires a $k = 10$ to recover to near 100% accuracy for the same data-set, which burdens the human user to disambiguate between 10 concrete examples of robot behavior before any robot behavior is performed.

We have reproduced the seq2seq NL-to-LTL language grounding model using the AllenNLP (Gardner et al., 2018) framework, with the same additive attention scheme. The new model has been trained using the same hyper-parameters and data-set reported in Danas et al. (2019)– except the number of epochs is significantly less, as we stop early with a patience of 5 epochs. We have also produced a new data-set, with a richer diversity of LTL formula complexity. This data-set has been sorted by target complexity, which we use to evaluate both models' ability to generalize to more complex LTL groundings that were not seen during training.

2 Goal Summary

The following is a list of goals we've had over the semester. Bold text indicates that the goal has been reached, and is detailed in this report. For goals not reached, we briefly describe the reason why it was not achieved below.

- **Collect the baseline data-set reported in (Oh et al., 2019). Acquire baseline model and results reported in Danas et al. (2019).**
- **Build and train the new NL-LTL language grounding model in AllenNLP, on baseline data-set and examine its performance.**
- **Improve new model over baseline model. Reduce the $k = 10$ beams and questions requirement from Danas et al. (2019).**
- **Analyze errors to figure out what kinds of scenarios our approach struggles with.**
- **Produce a new data-set, with more diverse LTL targets, that can be sorted by complexity for evaluating how well both models generalize within the top k beams.**
- **Evaluate how well both models generalize when they have yet to see LTL targets of a increasing complexity during training.**
- Augment the new model with constrained decoding and examine the performance impact.

Due to the difficulties producing and evaluating a new data-set, we present only a weak follow-up evaluation from our final presentation. We report the full failure of our second dataset in section 4. Also, the relevant AllenNLP repository currently hosts a half finished implementation of constrained decoding, and we did not have time to implement this in pytorch from scratch, in a way to fit in with our current language model.

3 Related work

We performed a literature review on methods of grounding natural language into formal language for commanding robots over the past decade, analysis of beam search decoding schemes, and methods used in our approach.

3.1 Language Grounding

We focus our discussion on the contributions by the historical group (Matuszek et al., 2013), the longest running group (Tellex et al., 2011; Howard et al., 2014; Kollar et al., 2017), and the state of the art group (Oh et al., 2019; Danas et al., 2019) in the respective paragraphs to follow.

Matuszek et al. (2013) talks about parsing natural language, in this case English, to Robot Control Language, a language as commands that can be understood by robots. The parsing here refers to conversion of text of English to RCL. RCL is a formal language defined by a grammar; and parsing is the process taking some input and forming an expression in that grammar. The input is natural language route instructions, and the parsing target is a statement in RCL that can be passed to a robot controller for planning or further disambiguation. Translation from NL to RCL occurs as such: first, the natural language command is parsed into a formal, procedural description representing the intent of the person. Then, the robot control commands are used by the executor along with the local world state to control the robot, thereby translating the NL commands into actions while exploring the environment. More specifically, the parsing is done by an extended version of unification based learner, whose grammatical formalism is based off a probabilistic version of combinatory categorial grammar. The authors provide a couple of clear examples of the RCL for the readers. For instance, "`current-loc: loc`" means the current position of the robot. During testing, the authors of the paper have attempted several inputs comprising both short and long inputs. For short inputs, the success rate of the robots completing the task given command in natural language is around 71.4 percent, whereas the success rate of long/complex input have a success rate 48.9%. Here, a success means the robot completes exactly the task given. There were many cases that robot were able to finish most of the tasks but failed to comply to one or two out of many tasks.

Tellex et al. (2011) proposes a graph-based probabilistic model that attempts to understand natural language commands for autonomous systems, allowing robots (in their case, forklifts) to navigate around their environment. This model first takes the given command and extracts a simple structured command. For example, `Put A on B` would be `EVENT(Put, OBJ(A), PLACE(on, OBJ(B)))`. It then induces a graph-based statistical model which they use to find the most probable sequence of actions. Although this model performs well with simple examples, it does not support negation (e.g. `don't go to A`), anaphora (e.g. `go to A and stay there`), and conditionals (e.g. `if you don't have the box yet, get it off the shelf`).

Howard et al. (2014) introduces a solution to solve how natural language interface for robot control to find the best sequence of actions that reflect the behavior intend by instruction. They presents a new model called the Distributed Correspondence Graph (DCG) to infer the most likely set of planning constraints from natural language instructions, and presents experimental results from comparative experiments that demonstrate improvements in efficiency in natural language understanding without loss of accuracy. This inspired us about how to build the natural language interface so that the model would perfectly understand the instructions. However, the language of the constraints was more engineering than formality, as pragmatism was of essence over theory.

Kollar et al. (2017) introduces to us a type of network that learns and predicts physical interpretation or groundings for linguistic constituents as opposed to traditional models where objects and commands were fixed phrases and did not possess the ability to learn. The network is known as Generalized Ground Graphs (G^3 for short). At inference time, the system is given a natural language command and infers the most probable set of groundings for the contextual environment and performs the task. This work is essentially the penultimate conclusion of the two previous works; it possesses both the strengths and weaknesses of both works. Since the work on G^3, the primary author of (Tellex et al., 2011) has moved onto LTL based language grounding and planning, as LTL is a constraint language that supports everything their thesis did not, and is much better suited to theoretical analysis.

Oh et al. (2019) introduces the novel approach called Abstract Product Markov Decision Process (AP-MDP) that seeks to handle both spatial abstraction as well as temporal ordering of events at once. The approach takes full advantage the formality of LTL for temporal ordering; previous language grounding work could not express concepts such as `avoid the red room while heading to the green room` and `go to the first floor then go to the landmark one`. Abstract Markov Decision Processes (AMDPs) define a natural hierarchical abstraction of relevant objects: that is all `landmarks` are in certain `rooms` that are on particular `floors`. Both the temporal ordering and object hierarchy are used within the AP-MDP to reduce resulting size of the concrete MDP (a product over the AMDP, the known universe of objects, and a particular LTL goal) as much as possible. This work enabled real-time robot interpretation of natural language commands, planning, and execution of behavior not seen in previous works to date.

3.2 Beam Search Decoding Schemes

Danas et al. (2019) exposes all k searched beams as output, instead of just the one most likely grounding. Then, their dialogue model produces a concrete example of robot behavior that canonically describes each of the k most likely groundings: canonical being robot behavior that achieves one of the grounded LTL goals and minimizes the other groundings that are also satisfied. Lastly, these examples are used to clarify the intended robot behavior for the originally uttered NL with the commanding human. This clarification is used to recover the LTL grounding intended by the original NL command. The authors perform a crowdsourced user study, confirming that untrained people can clarify intended robot behavior for $k = 10$ examples, given only an NL command. Thus, the language model only needs to produce the correct grounding in the top-k searched beams, as long as the human commanding the robot can disambiguate between k trajectories. However, producing these trajectories is quite expensive, and lowering $k = 10$ is paramount to real world feasibility.

Cho et al. (2014); Freitag and Al-Onaizan (2017) have analyzed the effectiveness of beam search. However, they do not seem to evaluate the top-k beams, and instead just focus on the top-1 decoding of resulting beam search. This is a reflection of natural language processing community's focus on certain machine translation tasks, as opposed to language grounding. When translating one natural language to another, BLEU similarity scores, and the marginal maximal likelihood of only the top-1 decoding is relevant.

Dyer et al. (2008) analyze decoding lattices in the context of traditional computational linguistics and natural language processing. This notion aligns with our top-k searched beams in the context of neural-network based natural language processing. However, it seems this line of work and appreciation of formal discrete structures has been abandoned with the promise and convenience of deepnet approaches. Similarly to Cho et al. (2014); Freitag and Al-Onaizan (2017), even their analysis of the lattice was limited by the current problems in natural language processing research at the time. As far as we know, no one has ever tried to use the beams within a question-answering protocol before Danas et al. (2019). The usefulness of the decoding lattice has only been elucidated via robotics: a rich intersection of natural language processing, formal methods, and machine learning.

3.3 Frameworks and Dataset Collection

Gardner et al. (2018) describes AllenNLP, a framework on top of PyTorch to easily design models for Natural Language Processing. Their approach to simplifying the process consists of implementing an API for efficient batching and padding, high-level implementations of common NLP subtasks, and changing the nature of writing code to be more modular, to allow easier changes to the parts of the model researchers want to target. This library served us well; it gave us a base to work on, as well as modules to tune for our research. That the project is open-source helped as well, as we could peek at the underlying code to diagnose any issues that arose.

Wang et al. (2015) a dataset gathering process for semantic parsing, summarized in Fig. 1. First, a builder creates a seed lexicon that contains the simplest mapping from natural language to a logical form. Then, a domain-general grammar is applied to more complex logical forms. Finally, crowdsourcing is used to create canonical natural language utterances for each logical form. We did not crowdsource; it was outside the scope of the project time and budget.

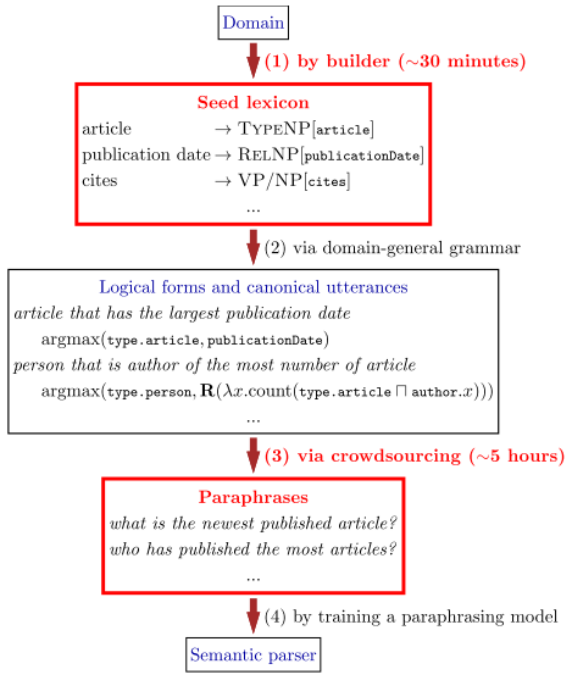

Figure 1: Process of gathering a dataset for a semantic parser, from (Wang et al., 2015).

4 Your dataset

Language grounding challenges the model output(s) to be grammatically correct, and match the commands intention. Both the LTL output syntax and semantics are incredibly precise, given an NL input space that is anything but. Command complexity presents another challenge for the model. LTL expressions can be viewed as an abstract syntax tree (AST), where each parent is an operand (`U`, `F`, ...) and every leaf is a terminal (`red_room`, `landmark_1`, ...). The depth of this tree is a decent measure of the command complexity. A model must be able to handle input commands reasonably more complex than what it has seen during training.

4.1 Baseline LTL Dataset

Our baseline dataset comes from (Oh et al., 2019) which includes 6,185 pairs of NL and LTL statements. The first example is: `navigate to the red room , always avoiding the blue room`– with a corresponding LTL expression of: `F ( red_room & ~ blue_room )`. These pairs are crowd-sourced from Mechanical Turk.

4.1.1 Properties and Statistics

This dataset contains expressions that can be organized into four broad sets. The first is expressions of the form `F ( TERM )`, where `TERM` is an arbitrary terminal. The second is expressions of the form `~(TERM) U (TERM)`. The third is of the form `F (TERM) & ~ TERM)`, and the final is expressions of the form `F (TERM) & OP (TERM)`, where `OP` was either `F` or `G`. Even though (Oh et al., 2019) reports a five-fold cross validation, it is not useful because the amount of repeated data will lead to overfitting, and underperformance on commands that do not fit into one of those categories. Certain LTL expressions were also included multiple times in the dataset, and the split for cross-validation was agnostic with regard to these duplicates. This means that the same expression could appear at both training and test time. We generated our own dataset to train on to correct these issues. Specifically, we sought a wider array of expression types, so that the model could learn from a more diverse set of expressions. We also wanted a dataset that was easily organized into different complexities, to evaluate the model's ability to generalize.

4.2 Complexity Sorted LTL Dataset

To test our model's ability to generalize, we generated an exhaustive list of LTL expressions that made use of one, two, and three operators, in combinations that do not result in unsatisfiability (there is no valid example behavior for such a command).

4.2.1 Operator Sub-spaces

The one-operator (one-op) space consists of three subspaces: F (`"eventually"`), G (`"global"`), and U (`"until"`). One such expression is `F ( red_room )`, since only one operator is used. The two-op space consists of a set of expressions denoted by two letters indicating the order in which operators were applied to generate it.

The two-op space includes FN (`"eventually"`, applied to `"not"`), FG, FA (`"eventually"`, applied to `"and"`), GN, GA, UN, UU, and UA. An example of an expression in the two-op space is `( ( red_room ) & ( first_floor ) ) U ( green_room )`, which is in the space because the `&` and `U` operators were applied.

The three-op space was generated in a similar fashion, and include FGN, FAA, FAN, GNA, GAA, UUU, UUN, UUA, UAA, UNN, UAN, UNA, AFF, AFG, and AGF. An example of an expression in three-op is `F ( ( first_floor ) & ( ( landmark_1`

) & (green_room))). With the addition of a new operator. There are 24 expressions in one-op, 252 in two-op, and 2276 in three-op, given the use of four terminals. Examples of a statement from each subspace can be found in Appendix A.

4.2.2 Data Pre-processing

We used a Python script (`ltlgen.py` in the repository) to generate a comprehensive list of LTL commands. For one-op commands, we simply create a set of the operator affixed to each terminal, if it is a unary operator, or enumerate every pair of terminals, if it is a binary operator. We do this recursively on the set of expressions in the two- and three-op space. We annotated each generated LTL expression with a generated NL counterpart. This NL command was generated by parsing[1] the AST for a given LTL expression, then by recursively applying string transformations into natural language.

This programmatic method of generating utterances resulted in broken English, e.g. `landmark one until the first floor and the red room until the red room`. We could not afford to crowdsource fixed NL utterances (Wang et al., 2015), and there were too many to do ourselves.

5 Baselines

Our baseline is the seq2seq model proposed in (Danas et al., 2019), but reimplemented in AllenNLP. Both English and LTL expressions were tokenized at the word level. We use an English embedding, LTL embedding, and one hidden layer, all with a dimensionality of 256. Finally, the decoder for the baseline is the AllenNLP `SimpleSeq2Seq` decoder, using a top-k beam search with a beam width of 10. This model was trained over 100 epochs, in accordance with the original model. This model uses additive attention as described in (Gardner et al., 2018), and uses Adam optimization.

5.1 Top-1 Decoding Accuracy

We calculated the percentage of LTL expressions generated by the model that matched their corresponding ground truth expressions using five-fold cross-validation. This method shuffles the dataset, trains on 80% of it (training set), then tests on the held out 20% (test set). This repeats for each fifth of the data, each having its own turn being held out. The results are reported in Table 1.

Fold	Old Model Acc.	New Model Acc.
1	79.61%	87.95%
2	82.90%	88.52%
3	82.41%	90.22%
4	82.70%	90.46%
5	82.11%	89.09%
Mean	81.95% $\pm$ 1.2%	89.64% $\pm$ 1.2%

Table 1: PyTorch (old) and AllenNLP (new) best-beam accuracies on the (Oh et al., 2019) model.

The new model outperforms the old one for every fold. We suppose that this is due to differences in implementation between the original PyTorch and the new AllenNLP models. Whereas the old model was created in a short time by people who did not specialize in natural language processing, the new implementation uses code from a specific NLP-based library, and is also implemented by a team with a basic knowledge in the area.

5.2 Top-k Beams Accuracy

We have reduced the need for the beam width to be $k = 10$ to $k = 3$. As seen in Figs. 2 and 3, we captured 99% accuracy decoding the original dataset in just three beams. If we expand the scope of the project back to the clarification dialogue, this means that we only need to ask the user for clarification three times, and we have a 99% chance of getting their original utterance clarified.

6 Your approach

We utilize the AllenNLP[2] library to implement this project, which is hosted on Google Colab[3]. This choice simplifies sharing the codebase, reduces the difference between running the code on different machines, and because Google has more powerful GPUs than the team. In addition, we used code we implemented ourselves from the `ltl-amdp`[4] repository. We were cynical about the performance relative to the baseline, as it had suspiciously high accuracies.

6.1 Evaluation Methodology

We evaluated our model on two accuracy methods. The first was on best-beam accuracy – the percent-

[1] We used the Lark parser, available here: `https://github.com/lark-parser/lark` on an LTL grammar defined by us (`ltl.lark` in the repo)

[2] `https:///github.com/allenai/allennlp`

[3] `https://colab.research.google.com/drive/1uJyr59tIkpELM4yCurVUXHnZM-12uuNU`

[4] `https:///github.com/cobielf/ltl-amdp/`

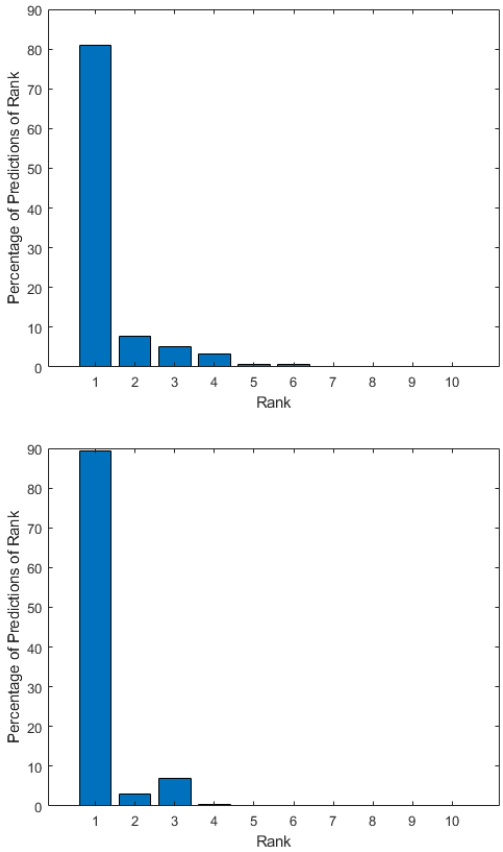

Figure 2: Frequency of correct groundings across all folds decoded at certain beam ranks (the x-th most likely beam), using the PyTorch model (top) and AllenNLP model (bottom).

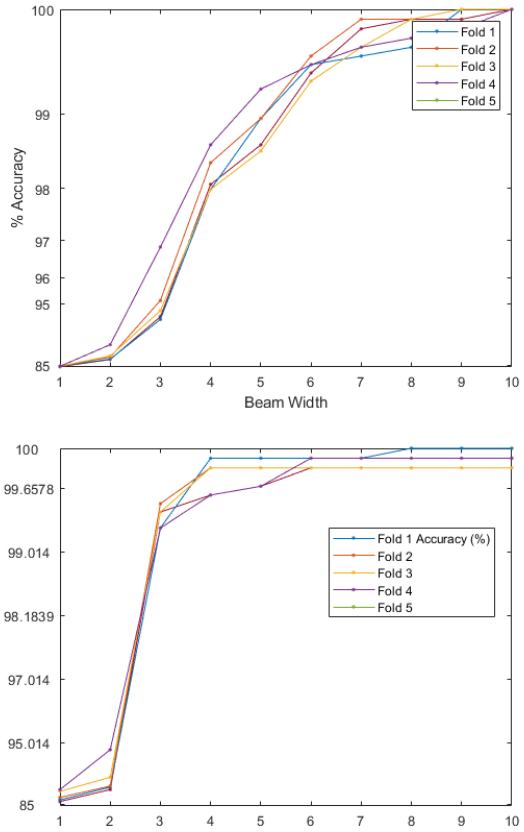

Figure 3: Percentage of correct decodings at beam width k or lower using the PyTorch model (top) and AllenNLP model (bottom).

age of predicted results that matched their corresponding ground truth expression, where we take the best beam at the end of the beam search. This is how beam search is usually used. The second was on in-top-k accuracy – the percentage of expressions where the correct ground-truth expression was in the final top-k beams.

6.2 Generalization Experiment

To begin with, we ran the `ltlgen.py` script, which enumerates all satisfiable one-op, two-op, and three-op expressions, and exports three tab-separated value (`.tsv`) files corresponding to the operator space. It also generates a training/test split, where the training set was one of the following, and the test set was the remainder of the dataset:

- 1. One-op and a random third of each two-op subspace.
  2. One-op and a random two-thirds of each two-op subspace.
  3. One-op and two-op.
  4. One-op, two-op, and a third of each three-op subspace.
  5. One-op, two-op, and two-thirds of each three-op subspace.
- One-op, two-op, and n-fifths of each three-op subspace, with $n \in [1:5)$
- One-op, two-op, and n-twenty-fifths of each three-op subspace, with $n \in [1:5)$
- One-op, two-op, and n statements of each three-op subspace, with $n \in [1:5)$

We then point the `DatasetReaders` in the Colab code to these datasets, then train and evaluate the model. The goal of this experiment was the opposite of what we might have traditionally wanted – we wanted to see if the experiment

caused new failures in the model, i.e. if introducing more complex expressions caused failures.

6.3 Results

To begin, we ran a validation using the first training/test split listed in the previous section. The results are in Table 2: Of particular interest to us was

Test Set	Best-Beam Acc.	Top-k Acc.
1	6.50%	6.99 %
2	3.68%	3.89 %
3	0%	0.703%
4	73.8%	99.9%
5	76.8%	100%

Table 2: Initial accuracy results based on op-space.

the third test set, where we exclusively trained on one- and two-op expressions, and tested on three-op expressions. The model performed extremely poorly here, so we created a new test/training split, which was the second bullet in the previous section. These experiment results, summarized in Table 3, are strongly indicative of a failure to generalize. The accuracy jumps from 0% to above 70%, just by introducing a few three-op expressions in the training dataset. While these results are worse

n heldout[5]	Best-Beam Acc.	Top-k Acc.
1	0.0%	0.615 %
2	72.2%	99.3 %
3	74.1%	100%
4	72.0%	94.74%
5	77.9%	99.2%

Table 3: Best-Beam and Top-k Accuracies for generalization experiment.

than the baseline model, they elucidate the shortcomings in its dataset and implementation, and explain why its results were so good. We can point to the homogeneity and relative simplicity of the (Oh et al., 2019) dataset as the reason for the baseline model's performance. Additional detail is given in Section 7.2.

To confirm our hypothesis, we tested the model on a continually-shrinking fraction of the three-op subspace (specifically, the last three mentioned in Section 6.2). The best-beam accuracy for this model is summarized in Fig. 4. Even with the smallest amount of three-op expressions in its dataset, the model can begin to make generalizations. From this, we conclude that the primary deciding factor in generalization is the number of operators in the training data expressions.

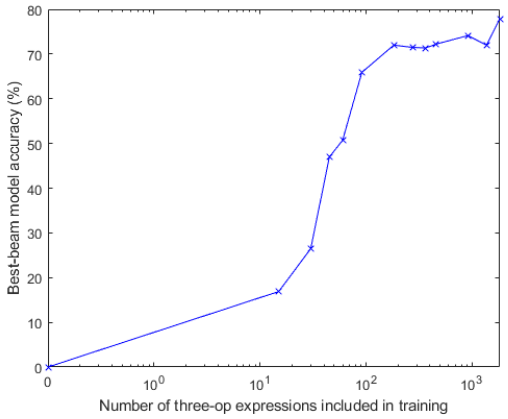

Figure 4: Increasing accuracy over more training data.

6.4 Issues

Working with AllenNLP and Colab presented their own quirks. Early on, we noticed that model accuracy increased monotonically across each fold of the cross-validation, with the final fold reaching a suspiciously high best-beam accuracy (99%!). We suspected that this was because we were actually training the same model across folds, meaning it was seeing most, if not all of the training data by the time it reached the final fold. This was solved at first by restarting the runtime between folds, then eventually by encapsulating the model in its own class to keep their memory segregated.

In addition, AllenNLP and Colab presented some problems with checkpointing. If the model was restored from a checkpoint, then the `Predictor` would not preserve the individual beam predictions, and only keep the top one. We suspect that this is a bug with either PyTorch or AllenNLP, but we decided to build around it since our models didn't take that long to train (about 5 minutes across each fold).

In practice, most implementations of beam search only take the best beam at the end of the search, and decode the rest. The documentation inside AllenNLP's beam search class even says so. Because this data was lost in its encapsulation, a significant amount of effort went into retrieving each beam after decoding was finished.

7 Error analysis

7.1 Baseline Failures

When the model fails, it produces grammatically correct LTL expressions– commonly with the incorrect terminals, rarely with wrong operands. For example, where the correct LTL expression would be `F ( green_room & F ( red_room ) )`, the model produces `F ( purple_room & F ( red_room ) )`. There is no particular class of terminal (i.e. rooms, floors, landmarks) that the model gets incorrect more than the others. An example of missed operands were `F ( yellow_room & F (landmark_1) )`, and the predicted expression was `~ ( landmark_1 ) U ( yellow_room )`, which is a semantically similar command interpretation.

The model almost never produces grammatically incorrect LTL expressions. This is either due to model over-fitting, or the fact that the baseline dataset has only four broad types of expressions. Since (Danas et al., 2019) includes the possibility of clarifying the original utterance, a goal is to see if the correct LTL expression ends up being one of those in the final k sets. Because this is a more relaxed constraint than just looking at the best beam, we get significantly higher percentages here (check Tables 1 and 3).

7.2 Failures Under Unseen LTL Complexity

One result of note, described with Table 3, is when we trained the model on one-op and two-op, and tested on three-op. From this, we can infer that **the model cannot generalize to expressions with more operators than it's seen during training**. In this case, the model does fail to render grammatically correct LTL sentences. One such example appears in our results: `((first_floor)U())U(` This is not a well-formed expression; the first `U` operator is missing an operand to its right, and the second `U` operator is also missing an operand to its right, for which it opened a parenthesis but never placed anything inside. The grammatical failures follow a pattern, however, in that the output expression becomes more nonsensical as it proceeds from left to right. This should not come as a surprise, as the model should create expressions in a lower operator space more easily. The model does not seem to fail in a specific way for certain types of expressions over the others. The only aspect that creates ungrammatical failures is the operator space.

8 Contributions of group members

- Cobi Finkelstein: Built and trained the model and contributed to writing report
- Harsha Malireddy: Many Initial (or simple) data collection and processing and contributed to writing report
- Chunghin Lee: Several Advanced (or complex) data collection and processing and contributed to writing report
- Jiaming Yuan: Error analysis and annotations and contributed to writing report
- Ray Huang: Researched and wrote about S.O.T.A. methods and described related works

9 Conclusion

Our primary takeaway from this project is that we believe that we've hit upon a rich point for further research. Language grounding and semantic parsing are relatively new topics, and we took a novel approach to top-k decoding and the information derived from executing it.

Among the difficulties of the project one of the hardest to overcome was the initial implementation of the project. Even though we made heavy use of frameworks, their use still took time to learn; in AllenNLP's case, we implemented experiments before we had taken a peek at it in class. Building around the quirks that showed up with the frameworks, and with Colab was also of note for its difficulty, as it occurred independently of our experiments, but still affected their outcomes.

We were surprised by the results. Particularly, we were surprised by how poor the (Oh et al., 2019) dataset was. We probably should not have been surprised, as natural language processing was not the domain for which the dataset was created; it was created in a robotics context. Creating a new dataset that was ordered in such a way as to be easily classifiable by hand, that also managed to split the test results so definitively, was an accomplishment we were particularly proud of.

The space for future work is promising. Most semantic parsing work parses down to some space of lambda calculus, so it will be interesting to see how adding a temporal element with LTL will pan out. Given significantly more time, we would have liked to implement constrained decoding for LTL

expressions, as we believe that it would have improved performance on unseen expression complexities.

References

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Danas, N., Finkelstein, C., Nelson, T., Krishnamurthi, S., and Tellex, S. (2019). Formal dialogue model for language grounding error recovery. In *Combining Learning and Reasoning – Towards Human-Level Robot Intelligence*. Robotics: Science and Systems.

Dyer, C., Muresan, S., and Resnik, P. (2008). Generalizing word lattice translation. Technical report, MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER STUDIES.

Freitag, M. and Al-Onaizan, Y. (2017). Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N., Peters, M., Schmitz, M., and Zettlemoyer, L. (2018). Allennlp: A deep semantic natural language processing platform. *arXiv preprint arXiv:1803.07640*.

Howard, T. M., Tellex, S., and Roy, N. (2014). A natural language planner interface for mobile manipulators. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6652–6659. IEEE.

Kollar, T., Tellex, S., Walter, M., Huang, A., Bachrach, A., Hemachandra, S., Brunskill, E., Banerjee, A., Roy, D., Teller, S., et al. (2017). Generalized grounding graphs: A probabilistic framework for understanding grounded commands. *arXiv preprint arXiv:1712.01097*.

Matuszek, C., Herbst, E., Zettlemoyer, L., and Fox, D. (2013). Learning to parse natural language commands to a robot control system. *Experimental Robotics Springer Tracts in Advanced Robotics*, page 403–415.

Oh, Y., Patel, R., Nguyen, T., Huang, B., Pavlick, E., and Tellex, S. (2019). Planning with state abstractions for non-markovian task specifications. *arXiv preprint arXiv:1905.12096*.

Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A. G., Teller, S., and Roy, N. (2011). Understanding natural language commands for robotic navigation and mobile manipulation. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Wang, Y., Berant, J., and Liang, P. (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)*, pages 1332–1342.

A Examples of expressions from each two-op and three-op subspace.

Space	Example Expression
FN	F (∼ (green_room))
FG	F (G (red_room))
FA	F ((landmark_1) & (green_room))
GN	G (∼ (landmark_1))
GA	G ((first_floor) & (landmark_1))
UN	(∼ (red_room)) U (red_room)
UU	((first_floor) U (landmark_1)) U (red_room)
UA	((green_room) & (red_room)) U (green_room)

Table 4: Two-op expression examples.

Space	Example Expression
FGN	F (G (∼ (red_room)))
FAA	F ((landmark_1) & ((green_room) & (landmark_1)))
FAN	F ((landmark_1) & (∼ (landmark_1)))
GNA	G ((∼ (landmark_1)) & (green_room))
GAA	G (((red_room) & (first_floor)) & (red_room))
UUU	(red_room) U (((first_floor) U (red_room)) U (green_room))
UUN	((∼ (first_floor)) U (green_room)) U (first_floor)
UUA	(landmark_1) U (((green_room) & (red_room)) U (first_floor))
UAA	((red_room) & (landmark_1)) U ((landmark_1) & (first_floor))
UNN	(∼ (landmark_1)) U (∼ (green_room))
UAN	((first_floor) & (green_room)) U (∼ (red_room))
UNA	(∼ (green_room)) U ((landmark_1) & (red_room))
AFF	(F (red_room)) & (F (green_room))
AFG	(F (red_room)) & (G (first_floor))
AGF	(G (first_floor)) & (F (landmark_1))

Table 5: Three-op expression examples.