Analysis of Tree-Structured Architectures for Code Generation

Samip Dahal **Adyasha Maharana** **Mohit Bansal**
Department of Computer Science
University of North Carolina at Chapel Hill
{sdpmas, adyasha, mbansal}@cs.unc.edu

Abstract

Code generation is the task of generating code snippets from input user specifications in natural language. Leveraging the linguistically-motivated hierarchical structure of the input can benefit code generation, especially since the specifications are complex sentences containing multiple variables and operations over various data structures. Moreover, recent advances in Transformer architectures have led to improved performance with tree-to-tree style generation for other seq2seq tasks e.g., machine translation. Hence, we present an empirical analysis of the significance of input parse trees for code generation. We run text-to-tree, linearized tree-to-tree, and structured tree-to-tree models, using constituency-based parse trees as input, where the target is Abstract Syntax Tree (AST) of the code. We evaluate our models on the Python-based code generation dataset CoNaLa and a semantic parsing dataset ATIS. We find that constituency trees encoded using a structure-aware model improve performance for both datasets. We also provide an analysis of those aspects of the input parse trees which are most impactful. For instance, we find that structure-aware encodings are better at modelling inputs with multiple variables and capturing long-range dependencies for code generation.[1]

1 Introduction

Code generation is the task of converting input user specifications written in natural language (NL) to code snippets in a target programming language. It is a task-driven variant of semantic parsing, which translates natural language input to formal machine-executable representation. Recent works have utilized the Abstract Syntax Tree (AST) - which is the syntactic tree representation of target source code - to generate better code snippets (Yin and Neubig, 2017, 2018; Sun et al., 2020; Rabinovich et al., 2017). The use of ASTs has achieved strong results but there has been relatively less work on utilizing the parse trees of the NL input. Constituency or dependency trees representing the syntactic structure of input can be leveraged to perform sub-tree alignment with corresponding AST of target code and benefit the downstream task. Hence, in this paper, we present several tree-to-tree models for the code generation task that convert the parse tree representation of NL input to AST representation of target source code. First, we base our model on the Transformer architecture (Vaswani et al., 2017). However, the standard Transformer is not designed to preserve the tree structure of the input parse trees. Hence, to better encode the trees, we modify a structure-aware Tree Transformer model (Nguyen et al., 2020) for the tree-to-tree code generation task. We focus on constituency-based parse trees in this paper because of space constraints as this is a short paper. Moreover, as pointed out by Nguyen et al. (2020), there is little evidence of constituency structures being learned implicitly in language models, whereas dependency structures have been shown to be implicitly embedded in models like BERT (Devlin et al., 2019; Hewitt and Manning, 2019). We evaluate our models on the CoNaLa dataset (Yin et al., 2018) and find that incorporating constituency parse trees in input using structure-aware encoders improves the quality of generated code. We further evaluate our models on the ATIS dataset (Hemphill et al., 1990), which translates natural language sentences into their lambda calculus logical forms and show that a structure-aware Transformer significantly improves performance over a standard Transformer.

We also focus on analyzing the input parse trees to find the aspects that benefit code generation. Our analysis comprises ablation experiments on our pro-

[1]Code available at: https://github.com/sdpmas/TreeCodeGen.

posed structure-aware model and pattern analysis of the output from different models with respect to the characteristics of input natural language specification. Specifically, we analyze the variation in performance with the presence of user-defined identifiers and variable entities in input sentence, and the complexity of input trees. We find that the structure-aware model improves performance when such identifiers and variables are present towards the end of the input sentences and when the input sentences are short in length.

2 Related Work

Code Generation. Code generation for general-purpose programming languages is a recent phenomenon, earlier works being focused on domain-specific languages (Gulwani and Marron, 2014; Raza et al., 2015). Recent works have mainly applied sequence-to-tree models for code generation, with the tree being the AST of target source code (Dong and Lapata, 2016; Yin and Neubig, 2017; Rabinovich et al., 2017; Yin and Neubig, 2018, 2019; Shin et al., 2019; Xu et al., 2020; Sun et al., 2020). While the use of ASTs for code generation has been substantially studied, to the best of our knowledge, the use of input parse tree for code generation is largely unexplored.

Semantic Parsing. Several methods have been proposed to parse natural language sentences to formal meaning representations like lambda calculus (Wong and Mooney, 2007), Alexa Meaning Representation Language (Kumar et al., 2017), Abstract Meaning Representations (AMR) (Banarescu et al., 2013), structured queries (Iyer et al., 2017; Yin and Neubig, 2018), etc. Many of the recent works for semantic parsing have focused on sequence-to-tree models leveraging tree structures like AST as the intermediate representation for target meaning representation (Yin and Neubig, 2018; Sun et al., 2020). Code generation can also be regarded as a form of semantic parsing where the target meaning representation is programming language snippet.

Source Trees and Structure-Aware Models. Several structure-aware tree-encoders have also been proposed to process the source trees (Chen et al., 2017a,b; Yang et al., 2017; Nguyen et al., 2020). While many of the tree-encoders are dependent on recurrent mechanism and hence are unparallelizable, Nguyen et al. (2020) propose a Transformer-based structure-aware model that is parallelizable. Concurrently, several tree-to-seq models have been proposed that leverage source syntactic trees for NLP tasks like machine translation (Eriguchi et al., 2016; Yang et al., 2017; Eriguchi et al., 2017; Chen et al., 2017b) and sentence modeling (Shi et al., 2018). There has been some work on leveraging hybrid tree - a joint tree-like representation of the NL sentence and corresponding meaning representation - for semantic parsing (Lu et al., 2008; Jie and Lu, 2018), while Harer et al. (2019) made use of source tree structures for code correction. However, the same is unexplored in the context of code generation. We study the use of tree-to-tree models for code generation and provide analysis of its various modules.

3 Our Models

3.1 Baseline (Sequence-to-Tree Model)

We use a standard Transformer model (Vaswani et al., 2017) as our natural language-to-code baseline. We build a sequence-to-tree model with a regular Transformer encoder and decoder. The encoder maps the source sequence $\mathbf{x} = x_1, x_2, ..., x_n$ to its vector representation $\hat{\mathbf{x}} = \hat{\mathbf{x_1}}, \hat{\mathbf{x_2}}, ..., \hat{\mathbf{x_n}}$, which is passed into the decoder. At each time step t, we linearize the AST generated till time step $t-1$ i.e. AST $y_{<t}$ and concatenate its embedding with the embedding of the corresponding parent actions, following Yin and Neubig (2017). Decoder takes this partial AST vector representation and source vector representation $\hat{\mathbf{x}}$ from encoder as input and expands the frontier non-terminal node of the partial AST. Here, ASTs are linearized by the pre-order depth-first traversal and the expansion of the AST, at each time step, is constrained by the grammar rules of the underlying programming language. We adopt the ASDL grammar and transition system (Yin and Neubig, 2018) that decomposes the production of an AST into a sequence of actions. At each time step t, the action a_t can be of 3 types (see Appendix for details). Given the input specification $\mathbf{x}$, the probability of generating an AST $\mathbf{y}$ can be expressed in terms of probabilities of generating corresponding actions: $p(\mathbf{y} \mid \mathbf{x}) = \prod_t p(a_t \mid \mathbf{x}, y_{<t})$. Here, a_t is the action at time step t and $y_{<t}$ is the partial AST generated upto time step t. We also use a pointer network (Vinyals et al., 2015) to allow the model to copy relevant entities from input sequence while generating a terminal AST node.

3.2 Linearized Tree-to-Tree Model

Here, we use the identical model architecture as our baseline (see Sec. 3.1) but we replace the input NL sequence with its linearized constituency-based parse tree. Constituency trees aim to describe syntactic structure of the sentence by dividing it into sub-phrases. As discussed in Sec. 1, this structural information can promote alignment between source and target sub-trees (AST), thereby improving downstream generation task. In our model, constituency trees are linearized by the pre-order depth-first traversal (see Fig. 5 in Appendix). Our output is the AST representation of code.

3.3 Structured Tree-to-Tree Model with Hierarchical Accumulation

A standard Transformer encoder (see Sec. 3.1) is not designed to process the structural information of input parse trees. On the other hand, many tree-based models have been proposed in the past to process the structural information (Chen et al., 2018; Eriguchi et al., 2016; Rao et al., 2019) but most of them are based on recurrent mechanism and hence, not parallelizable like Transformer-based models. This observation motivated us to build a Transformer-based structure-aware tree-to-tree model. In this paper, we adapt Tree Transformer, an attention-based tree-to-tree model with hierarchical accumulation proposed by Nguyen et al. (2020), for code generation. Hierarchical accumulation aims to encode the tree by performing a series of operations including upward cumulative-average and weighted aggregation on the interpolated tree matrix. Furthermore, the model includes hierarchical embeddings to induce biases that reflect hierarchy within each branch of the tree and among the siblings within a subtree. Finally, subtree masking is used to filter out irrelevant information during upward cumulative-average and weighted aggregation operations. In this model, our target is identical to that of our baseline i.e., the AST representation of the source code, which is later converted to source code with the help of the transition system. We linearize the AST in the same fashion as our baseline, concatenate it with the corresponding parent actions vector in the hidden dimension and pass it into the decoder along with the leaves and nodes vector representations from the encoder. We also add a pointer network (Vinyals et al., 2015) to allow the model to copy from leaves of input parse tree while generating a terminal AST node. Without pointer network, our model architecture is identical to the Tree Transformer, so we refer the reader to Nguyen et al. (2020) for a complete description of the model architecture.

Method	BLEU
Xu et al. (2020)	27.2
Baseline: Text-to-Tree	28.13
Model 1: Linearized Constituency Tree-to-Tree	27.71
Model 2: Structured Constituency Tree-to-Tree	**30.30**

Table 1: Results on the test set of CoNaLa dataset.

4 Experimental Setup

Dataset. We evaluate each of the models described in Sec. 3 on the CoNaLa (Yin et al., 2018) and ATIS (Hemphill et al., 1990) datasets. The CoNaLa dataset contains 2379 manually curated intent-snippet pairs for training (200 of which we use for validation) and 500 pairs for test. Although the CoNaLa dataset consists of 600k additional mined intent-snippet pairs, we train our models only on the manually-curated training dataset and compare them with Xu et al. (2020) model trained on the same dataset and without the use of reranker. The ATIS dataset consists of 4434, 491 and 448 pairs for training, validation, and test respectively. Following previous works, we use corpus-level BLEU-4 and exact-match accuracy metrics for evaluation on CoNaLa and ATIS datasets respectively. See Appendix for details on training and inference.

5 Results

Table 1 shows BLEU scores from our experiments on the CoNaLa dataset. Our baseline Transformer model outperforms previous state-of-the-art LSTM-based model (Xu et al., 2020) by 0.93 BLEU points. The linearized constituency tree-to-tree model hinders the BLEU score compared to our baseline. However, the structured constituency tree-to-tree model significantly outperforms baseline by 2.17 ($p<0.01$)[2] BLEU points and linearized constituency tree-to-tree model by 2.59 ($p<0.01$) BLEU points. It also outperforms the baseline model by 8% in terms of **human-evaluated** code quality (see Appendix). This suggests that the structured inputs can provide important cues for generating high quality code snippets through structure-aware encodings. This information is lost when trees are converted to linearized inputs, thereby

[2]Statistical significance is computed with 100K samples using bootstrap (Noreen, 1989; Tibshirani and Efron, 1993).

Method	Acc.
TRANX (Yin and Neubig, 2018)	86.2
TreeGen (Sun et al., 2020)	**89.1**
Baseline: Text-to-Tree	75.89
Model 1: Linearized Constituency Tree-to-Tree	53.57
Model 2: Structured Constituency Tree-to-Tree	86.83

Table 2: Results on the test set of ATIS dataset.

leading to a drop in performance over the text-to-tree baseline. It is important to note that our models have significantly higher number of parameters compared to (Xu et al., 2020) (roughly 44-49M for our models vs 2M for their model) as their model consists of only one layer of LSTM. However, we ran the LSTM model with higher number of parameters by increasing the embedding dimensions and the number of hidden layers in encoder LSTM and we did not see significant improvement in BLEU score. This indicates that the superior performance of our models is primarily due to the rather than the increased count of learnable parameters.

Table 2 shows accuracy scores from our experiments on the ATIS dataset. Our baseline model performs significantly worse than the LSTM-based TRANX (Yin and Neubig, 2018) and the accuracy further drops with the linearized constituency tree-to-tree model. Our structured model, however, performs significantly better than the aforementioned models ($p<0.01$), a trend we observed in results on the CoNaLa dataset as well. The accuracy of the structured model is still slightly worse than the TreeGen model (Sun et al., 2020). This might be because the TreeGen model consists of an AST reader which encodes the partial code tree generated in previous timesteps using structure-aware tree convolutions, during generation at each timestep. Our model lacks such a module for the target AST. Nonetheless, the overall trend among our three models suggests that parse trees benefit semantic parsing as long as their structure is incorporated in the model. However, if this extra hierarchical information is encoded in a linear fashion, it results in negative contribution to semantic parsing (row 4 in Table 2). Overall, our results also provide motivation for joint modelling of both, input and output parse trees, for semantic parsing.

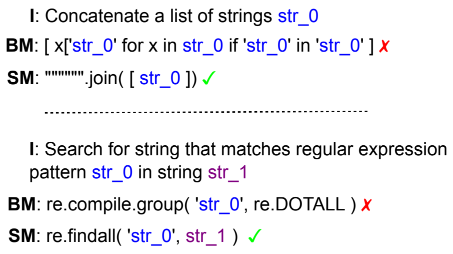
6 Analysis

6.1 Ablation Tests

We ablate our best model to understand the effect of the various modules in Tree Transformer on target task and present results in Table 3. First, we remove subtree masking which allows each node of the tree to attend over nodes that are not in the subtree rooted at that node in hierarchical accumulation. Second, we remove the use of hierarchical embeddings in our model. On the CoNaLa dataset, both experiments result in negative impact on the model's performance. This suggests that subtree masking is a crucial mechanism for structure-aware encoding i.e, for each node in the parse tree, only the relevant information within the subtree rooted at the node is useful. Comparatively, the results show that subtree masking is more important than hierarchical embeddings.

Method	CoNaLa
Structured Constituency Tree to Tree	**33.76**
- Subtree Masking	31.76
- Hierarchical Embeddings	32.62

Table 3: Ablation results for the structured constituency tree-to-tree model on the validation set of CoNaLa dataset. BLEU-4 metric is used to evaluate predictions.



```
    I: Concatenate a list of strings str_0
BM: [ x['str_0' for x in str_0 if 'str_0' in 'str_0' ] ✗
SM: """""""".join( [ str_0 ]) ✓
    --------------------------------------------------------
    I: Search for string that matches regular expression
    pattern str_0 in string str_1
BM: re.compile.group( 'str_0', re.DOTALL ) ✗
SM: re.findall( 'str_0', str_1 ) ✓
```

Figure 1: Sample predictions of our models when quoted string(s) appear towards the end of input sentence. These strings (highlighted) are replaced by placeholders. (I=Input, BM=Baseline Model, SM=Structured Model)

6.2 Pattern Analysis

Following Yin and Neubig (2017) and Xu et al. (2020), we next analyze the input intents and the corresponding code generated by the baseline model and the structured model (on a subset of test samples of CoNaLa datset) to find recurring patterns. First, we observe that input specifications in CoNaLa dataset contain quoted strings, which often occur as user-defined identifiers or strings in generated code as well. We find that when these quoted strings appear towards the end of the input sentence, the difference in quality of output code by the two models in terms of average BLEU score is higher than usual i.e., more than 5 BLEU points (row 2 of Table 4). Moreover, when the input sentence contains two or more quoted strings,

I: Create list var_0 containing 100 instances of object var_1
BM: var_0 = [(var_1 + var_0) for var_1, in 100(var_0)] ✗
SM: var_0 = [var_1() for _ in range(100)] ✓

....

I: Convert string var_0 into a list of integers var_1
BM: var_1 = [int(x) for x in 'var_1'.split(var_0)] ✗
SM: [int(i) for i in var_0.split()] ✓

Figure 2: Sample predictions of our models when multiple quoted strings appear in the input sentence. (I=Input, BM=Baseline Model, SM=Structured Model)

Pattern	**Baseline**	**Structured**
All Intents	20.23	**23.74**
Ending with quoted string	20.79	**26.42**
Multiple quoted strings	23.55	**29.20**
No quoted strings	10.04	**12.42**

Table 4: Comparison of average BLEU scores of baseline model and structured model in relation to different characteristics of input intents of CoNaLa dataset. Results are shown on the test set.

the baseline model often fails to capture the semantic relationship between those strings in the output code resulting in lower BLEU scores (row 3 of Table 4). However, in the absence of any quoted strings, the structure-aware model does better than the baseline by only 2 BLEU points (row 4 of Table 4). This shows that the structured input, when paired with a structure-aware encoder, helps capture dependencies between semantic units. Fig. 1 and Fig. 2 provide examples of both these scenarios and Table 4 compares the average BLEU scores.

Similarly, we notice that there are variable entities like city, airline, airport, time, etc. in the input specifications, which also appear in the corresponding outputs in the ATIS dataset. We find that our structure-aware model outperforms the baseline model by 12.57 points when such variables occur at the end of the input sentence (see row 3 in Table 6), suggesting that the model is able to capture long-term dependencies (see Fig. 8).

6.3 Comparison Based on Input Complexity

We compare the performance of the baseline text-to-tree and structured tree-to-tree models w.r.t. input complexity i.e. the length of input sentences and height of the input parse trees in the test set of CoNaLa dataset. The variation of mean BLEU scores w.r.t. length of input sentence and height of input trees is shown in Figures 3 and 4 respectively. In both figures, we observe that the structure-aware model outperforms baseline by wider margins for inputs of shorter length and height. Similarly, there are smaller but consistent improvements for inputs of medium complexity. The margins are largest for samples of high complexity, but this observation is supported by relatively few data points (see scatter plots in Appendix). From these results, we infer that the structured model is particularly helpful for short input sentences or parse trees in code generation. Similarly, the structured model significantly outperforms the baseline for shorter intent lengths in the ATIS dataset. However, we did not find any clear linkage between the height of input tree and the performance of our models on the ATIS dataset (see Figures 9 and 10 in Appendix).

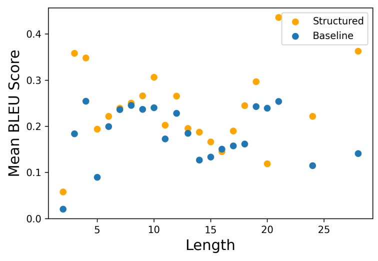


Figure 3: Plot of input intent length vs. mean BLEU score for our baseline and structured model on the test set of CoNaLa dataset.

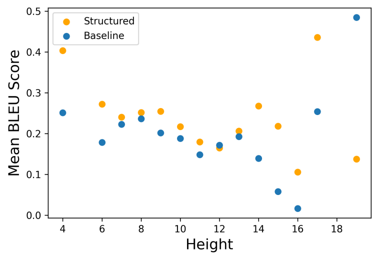


Figure 4: Plot of height of input parse tree vs. mean BLEU score for our baseline and structured model on the test set of CoNaLa dataset.

7 Conclusion

We experimented with models to utilize input constituency parse trees for code generation and semantic parsing. Our tree-to-tree model significantly outperforms other approaches for code generation and is competitive for semantic parsing. We find that the hierarchical structure of parse trees helps the structure-aware model capture semantic relationships between user-defined identifiers and variable entities in the input intent.

Acknowledgments

We thank Hyounghun Kim and the reviewers for their useful feedback. This work was supported by NSF-CAREER Award 1846185 and a Microsoft Investigator Fellowship.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Huadong Chen, Shujian Huang, David Chiang, and Jiajun Chen. 2017a. Improved neural machine translation with a syntax-aware encoder and decoder. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1936–1945, Vancouver, Canada. Association for Computational Linguistics.

Kehai Chen, Rui Wang, Masao Utiyama, Lemao Liu, Akihiro Tamura, Eiichiro Sumita, and Tiejun Zhao. 2017b. Neural machine translation with source dependency representation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2846–2852, Copenhagen, Denmark. Association for Computational Linguistics.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA. Curran Associates Inc.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833, Berlin, Germany. Association for Computational Linguistics.

Akiko Eriguchi, Yoshimasa Tsuruoka, and Kyunghyun Cho. 2017. Learning to parse and translate improves neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 72–78, Vancouver, Canada. Association for Computational Linguistics.

Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The ATIS spoken language systems pilot corpus. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27,1990*.

John Hewitt and Christopher D Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138.

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

Zhanming Jie and Wei Lu. 2018. Dependency-based hybrid trees for semantic parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2431–2441, Brussels, Belgium. Association for Computational Linguistics.

Anjishnu Kumar, Arpit Gupta, Julian Chan, Sam Tucker, Bjorn Hoffmeister, Markus Dreyer, Stanislav Peshterliev, Ankur Gandhe, Denis Filiminov, Ariya Rastrow, et al. 2017. Just ask: Building an architecture for extensible self-service spoken language understanding. In *1st Workshop on Conversational AI at the Conference on Advances in Neural Information Processing Systems (NIPS) 2017*.

Wei Lu, Hwee Tou Ng, Wee Sun Lee, and Luke Zettlemoyer. 2008. A generative model for parsing natural language to meaning representations. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 783–792.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. 2020. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*.

Eric W Noreen. 1989. *Computer-intensive methods for testing hypotheses*. Wiley New York.

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

Jinfeng Rao, Kartikeya Upasani, Anusha Balakrishnan, Michael White, Anuj Kumar, and Rajen Subba. 2019. A tree-to-sequence model for neural NLG in task-oriented dialog. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 95–100, Tokyo, Japan. Association for Computational Linguistics.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Haoyue Shi, Hao Zhou, Jiaze Chen, and Lei Li. 2018. On tree-based neural sentence modeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4631–4641, Brussels, Belgium. Association for Computational Linguistics.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *AAAI*, pages 8984–8991.

Robert J Tibshirani and Bradley Efron. 1993. An introduction to the bootstrap. *Monographs on statistics and applied probability*, 57:1–436.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28, pages 2692–2700. Curran Associates, Inc.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967, Prague, Czech Republic. Association for Computational Linguistics.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online. Association for Computational Linguistics.

Baosong Yang, Derek F. Wong, Tong Xiao, Lidia S. Chao, and Jingbo Zhu. 2017. Towards bidirectional hierarchical representations for attention-based neural machine translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1432–1441, Copenhagen, Denmark. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

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

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559.

Appendices

A Our Models

A.1 Baseline (Sequence-to-Tree Model)

We use a standard Transformer model (Vaswani et al., 2017) as our natural language-to-code baseline. We build a sequence-to-tree model with a regular Transformer encoder and decoder. The encoder maps the source sequence $\mathbf{x} = x_1, x_2, ..., x_n$ to its vector representation $\hat{\mathbf{x}} = \hat{\mathbf{x_1}}, \hat{\mathbf{x_2}}, ..., \hat{\mathbf{x_n}}$, which is passed into the decoder. At each time step t, we linearize the AST generated till time step $t-1$ i.e. AST $y_{<t}$ and concatenate its embedding with the embedding of the corresponding parent actions, following Yin and Neubig (2017). The decoder takes this partial AST vector representation and the source vector representation $\hat{\mathbf{x}}$ from encoder as input and expands the frontier non-terminal node of the partial AST. Here, the ASTs are linearized by the pre-order depth-first traversal and the expansion of the AST, at each time step, is constrained by the grammar rules of the underlying programming language. We adopt the ASDL grammar for Python and transition system (Yin and Neubig, 2018) that decomposes the production of an AST into a sequence of actions. At each time step t, the action a_t can be of 3 types:

- *ApplyRule Action*: Applies production rule R to the partial AST.
- *Reduce Action*: Denotes the completion of a field with optional or multiple cardinalities.
- *GenToken Action*: Expands a terminal node by generating a leaf token.

Given the input specification $\mathbf{x}$, the probability of generating an AST $\mathbf{y}$ can be expressed in terms of the probabilities of generating the corresponding actions:

$$p(\mathbf{y} \mid \mathbf{x}) = \prod_t p(a_t \mid \mathbf{x}, y_{<t}) \quad (1)$$

Here, a_t is the action at time step t and $y_{<t}$ is the partial AST generated upto time step t. We also use a pointer network (Vinyals et al., 2015) to allow the model to copy relevant entities from input sequence while generating a terminal AST node with GenToken Action.

B Experimental Setup

Training and Inference. All of our models have 6 encoder layers and 6 decoder layers. Our models are trained on GPUs using Google Colab and each model takes 2-3 hours for a single run. We perform manual hyperparameter tuning, using 4-5 runs for each model. We tried learning rates within the range [1e-4, 5e-5]. After manual tuning, for CoNaLa dataset, we trained all the models using learning rate of 1e-4. For ATIS dataset, we use learning rate of 2e-4 for our baseline model, 5e-5 for our Linearized Tree-to-Tree model and 4e-5 for our structured model. We use batch size of 64.

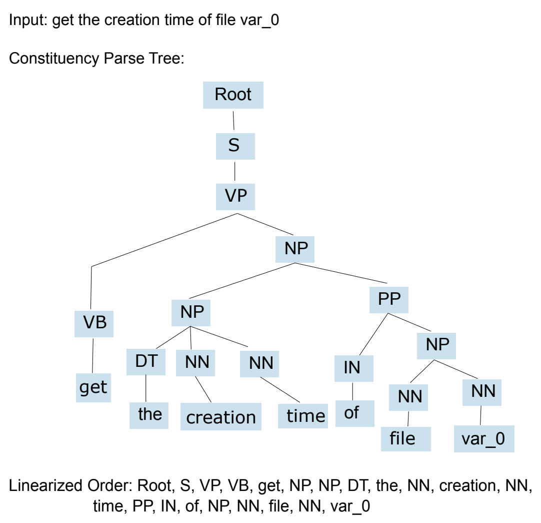

Input: get the creation time of file var_0
Constituency Parse Tree:
Root
S
VP
NP
VB
NP
PP
get
DT
NN
NN
IN
NP
the
creation
time
of
NN
NN
file
var_0
Linearized Order: Root, S, VP, VB, get, NP, NP, DT, the, NN, creation, NN, time, PP, IN, of, NP, NN, file, NN, var_0

Figure 5: Constituency parse tree of natural language specification *get the creation time of file var_0* and its linearized form. Words from the input specification are leaves of the tree.

Models	Wins	Loses	Tie
Structured T2T vs. Baseline	35%	27%	38%

Table 5: Results from human evaluation of generated code. Wins and Loses refer to the %times code generated from structured tree-to-tree model was chosen over those from baseline model.

We parse source text into constituency trees using Stanford CoreNLP parser (Manning et al., 2014). During inference, we use beam search with beam size of 30 for CoNaLa dataset and beam size of 1 for ATIS dataset to predict the output AST for a given natural language intent. We begin the beam search with one AST initialized with the root node and run until maximum time-step T or until we find K complete ASTs, where K is the beam-size. The maximum time-step is set to 200.

C Results

C.1 Human Evaluation

We also perform human evaluation of 100 samples from the CoNaLa dataset (see Table 5). The annotator (non-coauthor graduate student, proficient in Python) was instructed to pick the better code output for a given input specification. The samples contained shuffled outputs from our baseline and structure-aware models. Outputs from structure-aware model were preferred 35% of the times while those from the baseline were preferred 27% of the times and rest of the instances ended in a tie over code quality.

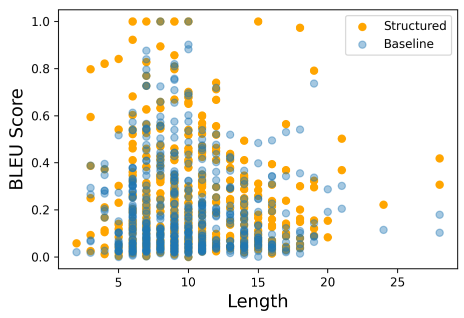


Figure 6: Scatter plot of input length vs. BLEU score for samples from the test set of CoNaLa dataset.

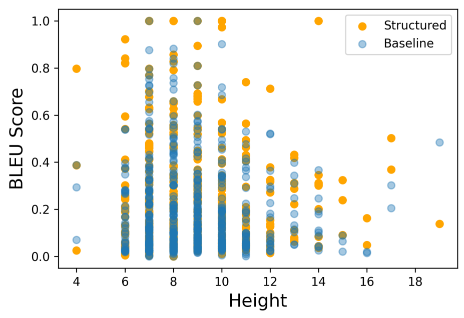


Figure 7: Scatter plot of height of input parse tree vs. BLEU score for samples from the test set of CoNaLa dataset.

C.2 Scatter Plots

We present two scatter plots for demonstrating the effect of input complexity on model performance for the CoNaLa dataset. Fig. 6 compares sentence-level BLEU score of predictions from our baseline and structured models against the length of input sentences. Similarly, Fig. 7 compares sentence-level BLEU score of predictions from our baseline and structured models against the height of input parse trees. Both of these comparisons are performed on the test set of CoNaLa dataset.

C.3 Pattern Analysis

Following Yin and Neubig (2017) and Xu et al. (2020), we next analyze the input intents and the corresponding code generated by the baseline model and our best model (on a small test sample), i.e., Structured Constituency Tree-to-Tree to find recurring patterns. We observe that the code generated by the structured model is significantly better for input intents containing certain characteristics. First, we observe that input specifications in CoNaLa dataset contain quoted strings (see placeholder `str_0` in Fig. 1 of the main text). These

I: what airport is at ci0
BM: (lambda $0 e (and (airport $0) (loc:t $0) (to $0 ci0))) ✗
SM: (lambda $0 e (and (airport $0) (loc:t $0 ci0))) ✓

....

I: look for a flight to ci0
BM: (lambda $0 e (and (flight $0) (to $0 ci0) (from $0 ci0))) ✗
SM: (lambda $0 e (and (flight $0) (to $0 ci0))) ✓

Figure 8: Outputs of our baseline and structured model for the ATIS dataset when variable entities appear at the end of input sentences.

Pattern	**Baseline**	**Structured**
All Intents	75.89	**86.83**
Not ending with a variable	74.52	**80.18**
Ending with a variable	76.31	**88.88**

Table 6: Comparison of accuracies of baseline model and structured model in relation to different characteristics of input intents of ATIS dataset. Results are shown on test set.

strings often occur as user-defined identifiers in the input sentence and as strings in generated code as well. We find that when these quoted strings appear towards the end of the input sentence, the difference in quality of output code by the two models in terms of average BLEU score is higher than usual i.e., more than 5 BLEU points (see row 2 in Table 4). We also find that when the input sentence contains two or more quoted strings, the baseline model often fails to capture the semantic relationship between those strings in the output code, resulting in lower BLEU scores. However, the structure-aware model succeeds at the task, resulting in higher BLEU scores (see row 3 in Table 4). In the absence of any quoted strings, the structure-aware model does better than the baseline by only 2 BLEU points (see row 4 in Table 4). This shows that when the structured input is paired with a structure-aware encoder, it helps capture the semantic relationships between multiple units. Fig. 1 and Fig. 2 provide examples of both these scenarios and Table 4 compares the average BLEU score of all the examples in the test set with 1) quoted string at the end of input sentence, 2) two or more quoted strings and 3) zero quoted strings,

Similarly, we notice that there are variable entities like city (`ci0` in Fig. 8), airline, airport, time, etc in the input specifications, which also appear in the corresponding outputs in the ATIS dataset. Such variables are anonymized with identifiers of same type following (Dong and Lapata, 2016). We find that when such variables occur at the end of the input sentence, our structured model does signifi-

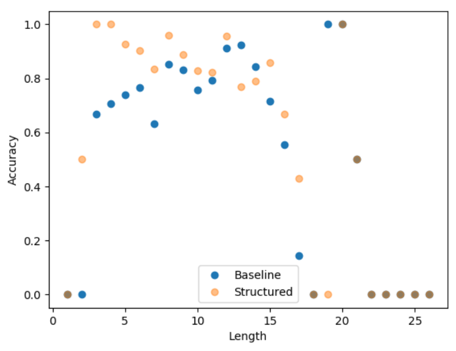


Figure 9: Plot of length of inputs vs accuracy on the test set of ATIS dataset.

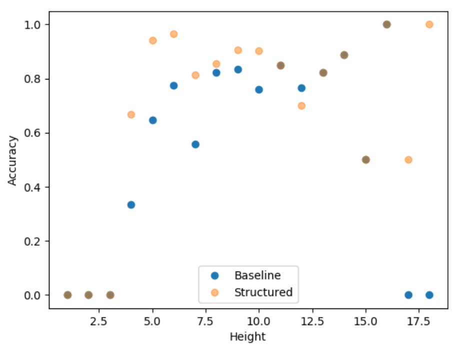


Figure 10: Plot of height of input parse trees vs accuracy on the test set of ATIS dataset.

cantly better than our baseline (see row 3 in Table 6) but the difference decreases in cases where the variables don't occur at the end of the input sentence (see row 2 in Table 6). Fig. 8 provides examples of the cases where variable entities occur at the end of the input sentences.

C.4 Comparison Based on Input Complexity

We compare the performance of our two models with respect to the complexity of input sentences. We rank the complexity of an input sentence by its length and the height of the corresponding parse tree i.e., the longest length of the path from the root node of the tree to its leaves. Firstly, we do this analysis on CoNaLa dataset. Fig. 3 presents a plot of length of input sentences and mean BLEU scores of generated code snippets. Fig. 4 presents a plot of height of input trees and mean BLEU scores of generated code snippets. Similarly, Figures 6 and 7 are scatter plots of sentence-level BLEU scores for generated code snipppets vs. length of input sentences and height of input parse trees respectively. We can see in both Fig. 3 and 4 that there is a wider gap between mean BLEU score of our structured model and baseline in the beginning, with the structured model performing significantly better. The gap narrows in the middle and widens towards the end. However, as we can see from the Figures 6 and 7, there are very few data points towards the end to draw any conclusion. From these observations, we infer that for code generation, structured model is particularly helpful for short input sentences or for short input parse trees. Similarly, the structured model significantly outperforms the baseline for shorter intent lengths in the ATIS dataset. However, we did not find any clear linkage between the height of input sentences and performance of our models for semantic parsing with the ATIS dataset. Fig 9 plots length of input sentences against accuracy and Fig 10 plots height of input parse trees against accuracy of outputs for samples in the test set of ATIS dataset.