The Comparison of Encoder Performance of NL2Code Generation with AST

Kehang Zhang[1], Shuangyi Tan[1], Zifan Wang[1], Qianyi Deng[2]
Department of Computer Science[1], Department of Statistical Science[2]
University College London
{ucabkz4,ucabst4,ucabzw9,ucakqde}@ucl.ac.uk

Abstract

As a challenging yet useful task, automated code generation has remained attractive to NLP researchers. In this paper, we develop a model that parses natural language intents to snippets of Python code.

To enforce the syntactic coherence of the generated code, we adopt the Abstract Syntax Tree method to regularize code generation. Particularly, we use the state-of-the-art TRANX model as the baseline and compare the performances of different architectures by replacing the original Bi-LSTM encoder in TRANX with GRU or a purely attentional encoder layer. Our experiment shows that the modified models have significantly faster training speed while maintaining comparable performances as the baseline.

The code that used to train and evaluate our models is available on https://github.com/kzCassie/ucl_nlp.

1 Introduction

The code generation task has attracted many NLP researchers over the years due to its challenging nature and practical importance. With good code generation tools, IT practitioners could greatly enhance their productivity by making fewer coding mistakes or automating the completion of duplicate requests. With proper design, code generation tools can also help a novice programmer efficiently get acquainted with a new programming language at ease.

Attempts with code generation have been made in various forms ranging from IDE integrated auto-completion tools to Macro recording functionalities based on user-interface interactions (e.x Microsoft Office bundle). In particular, this project focuses on a special use case where the user inputs a natural language utterance describing the task he/she wishes to accomplish and the system produces a code snippet that meets the requirement. This project adopts a neural approach to address this task.

Relevant methods exist in the field, yet many code generation problems remain unsolved. This is because, unlike general seq2seq natural language tasks, the outputted code sequence should not only meet the semantic requirements but also maintain a certain syntactic structure to be executable. However, due to the uncontrollable nature of neural networks, such goals can hardly be achieved directly.

TRANX is a transition-based neural code generator that cleverly tackles this problem by converting the original code generation task to the task of building an Abstract Syntax Tree (AST) via predicting a series of tree-building 'Actions'. Each AST can later be parsed into a code snippet. Moreover, TRANX predicts Actions in a language-agnostic way so that the same model can be easily adapted to a different coding language as long as appropriate pre-processing and after processing steps are conducted. Such versatility is advantageous considering the various coding languages and packages existing in the market. TRANX has achieved state-of-the-art results on many different code-generation datasets.

However, when examining the results of TRANX, we also identified some issues. First, TRANX uses a standard Bi-directional Long Short-Term Memory (LSTM) network as the encoder and the decoder. Such recurrent structure decides that it has high complexity and is computationally costly when dealing with long sentences. Second, although LSTM deals with memory loss to some extent, the current code generator still suffers from learning dependencies between positions that are further apart.

As an attempt to address these problems, this project explores alternative network architec-

tures by replacing the original Bi-LSTM encoder adopted by TRANX with different encoders such as Gated Recurrent Units (GRUs) and attentional encoder. Both alternatives are more computationally efficient while also achieving comparable performances. While the original TRANX model achieves a corpus BLEU of 0.30, the two alternatives achieve a corpus BLEU around 0.28, with TRANX_GRU cutting down the training time by 29.8% and the TRANX_attentional_encoder cutting down the time by 34.2% per epoch.

2 Related Works

Efforts of automatic code generation date back to 1969, when Waldinger and LEE uses algorithmic and mathematical approaches to generate LISP programs - the second-oldest high-level programming language in the world. There are also some other attempts conducted in this field including generating code from a laid-out table (Graham, 1980) or from the Unified Modeling Language (Usman and Nadeem, 2009). However, all of these approaches require the user's intentions to be expressed in certain logical forms in the first place, which causes a lack of flexibility.

Recent developments in neural-based NLP techniques have brought a lot of new possibilities to this task. Naively, we can treat code as a special type of 'natural language' and outstanding seq2seq models such as Neural Machine Translation (NMT) models (Barone and Sennrich, 2017) and the self-attention based transformer (Kusupati and Ailavarapu) model can be directly applied. However, such approaches ignore the special syntactic requirements of the code generation task and we need to find ways to constrain the generation process.

In this paper, we uses the Abstract Syntax Tree (AST) (Yin and Neubig, 2017) to guide the code generation process. The AST is a domain-dependent tree structure that can be constructed following the domain-independent Abstract Syntax Description Language (ASDL) grammar (Wang et al., 1997). Relevant works adopting this thread include Abstract Syntax Networks (ASNs) (Rabinovich et al., 2017), tree-structural language modeling (SLM) (Alon et al., 2020) and TRANsition-based abstract syntaX parser (TRANX) (Yin and Neubig, 2018), among which, the TRANX model will serve as a baseline for this project.

Along with the rapid development of neural methods, the construction and publication of NL2code datasets such as Django (Oda et al., 2015), StaQC (Yao et al., 2018) and CoNaLa (Yin et al., 2018) have also greatly enhanced the development in solving this task. In this project, we will use the CoNaLa dataset for various experiments.

3 Methodology

During training time, instead of training a seq2seq model directly from the original code snippets, the AST approach first parses snippets to sequences of 'actions', which can then be learned by a seq2seq model. During inference time, we first predict sequences of actions using the trained model, from which code snippets can then be reconstructed following the reverse steps accordingly.

In this section, we first explain the pipeline of generating sequences of actions from code snippets. Second, we introduce the original TRANX encoder-decoder architecture in details, as well as our modifications on the encoder part.

3.1 Transformation from Code to Actions

3.1.1 Code to Python AST

Given a Python snippet, it is first converted into a Python AST using the astor package.

3.1.2 Python AST to ASDL AST

Abstract Syntax Description Language (ASDL) is a grammar that describes the abstract syntax of compiler intermediate representations and other tree-like data structures. A Python AST is essentially still language-dependent. By using the predefined Python ASDL grammar, the domain-dependent Python AST can be converted to a general-purpose ASDL AST which is independent of the coding language.

3.1.3 ASDL AST to Actions

The ASDL grammar consists of five components: primitive type, composite type, constructor, composite node, and field. The creation of an ASDL AST strictly follows the transition system defined in the ASDL grammar. The workflow in Figure 1 exemplifies the construction process of an ASDL AST given a sequence of actions. During training time, the reverse process is conducted, i.e. we convert the ASDL AST to a sequence of tree-building actions.

During the growth of an AST, Three types of actions are involved.

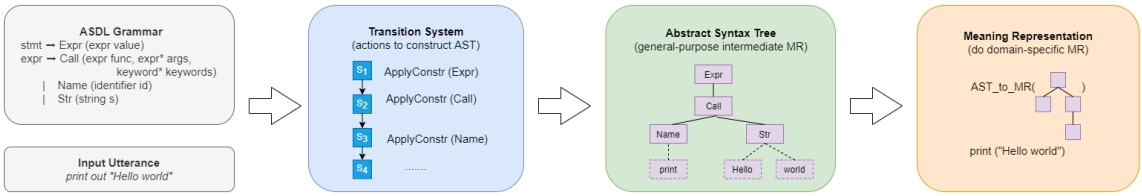

Figure 1: The workflow for model

- **APPLYCONSTR**: This action aims to expand the tree frontier by adding a node.
- **REDUCE**: This action indicates the completion of a branch of the tree and the pointer should be reduced to the appropriate ancestor node where the new frontier lies.
- **GENTOKEN**: This action generates a token of the primitive type.

Figure 2 shows a simple example of an AST that is constructed from the code snippet *"print('Hello World')"*. The tree is derived and executed in a depth-first manner and the time steps are marked in the figure. The red nodes correspond to the APPLYCONSTR actions whereas the grey nodes correspond to the GENTOKEN actions.

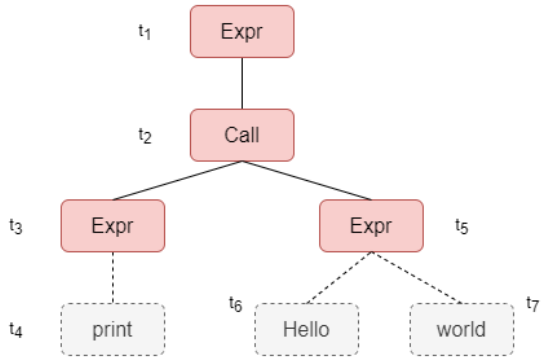

Figure 2: The AST of the Python code snippet *"print('Hello World')"*, where t_i marks the action executed at time step i.

3.2 Encoder

With a given natural language utterance x, the probability of an AST is equal to the probability of generating the corresponding sequence of actions. Using $\mathbf{z}$ to denote the AST, and a_t to denote the action taken at time step t, we have:

$$p(\mathbf{z}|\mathbf{x}) = \Pi_t p(a_t|a_{<t}, \mathbf{x}).$$

The original TRANX model uses Bi-LSTM as the encoder. In our project, we replace the encoder with GRU and attentional encoder and analyse the differences.

3.2.1 TRANX

TRANX uses Bi-directional LSTM both as the encoder and the decoder. The encoder takes-in a sequence of natural language tokens $\{x_i\}_{i=1}^n$ and produces hidden states $\{h_i\}_{i=1}^n$ while maintaining cell states $\{c_i\}_{i=1}^n$. The hidden states are then passed into a linear layer upon which an attention mechanism is used to generate query vectors. The last cell state c_T produced by the encoder is used to initialize the first LSTM cell of the decoder.

After getting the query vectors, an additional linear layer and the Soft-max function are applied to generate probabilities of subsequent actions. The TRANX network architecture is shown in Figure 3 where the purple boxes mark the Bi-LSTM encoder and decoder and the red vectors represent the query vectors generated via the attention mechanism.

3.2.2 GRU Encoder

GRU is a variant of LSTM, which is firstly proposed by Cho et al. (2014). The difference is that LSTM has three gates (a forget gate, an input gate, and an output gate) that control the information dissemination and forgetting, while GRU only conserves two gates: reset gate and update gate. The reset gate directly acts on the hidden state and decides the reservation of the previous state, while the update gate selects which hidden state should be changed and controls the information flow. Although the cell state c_t is abnegated in GRU, the same information is contained in the hidden state which can be used to initialize the decoder.

Generally, GRU has a similar structure as LSTM and has shown similar performance in different tasks. However, with the reduction of components, GRU requires less computation time. On the basis of the encoder-decoder structure suggested by TRANX, one attempt we conducted is to change the original Bi-LSTM encoder to GRU. The modified part of the original architecture is shown in the

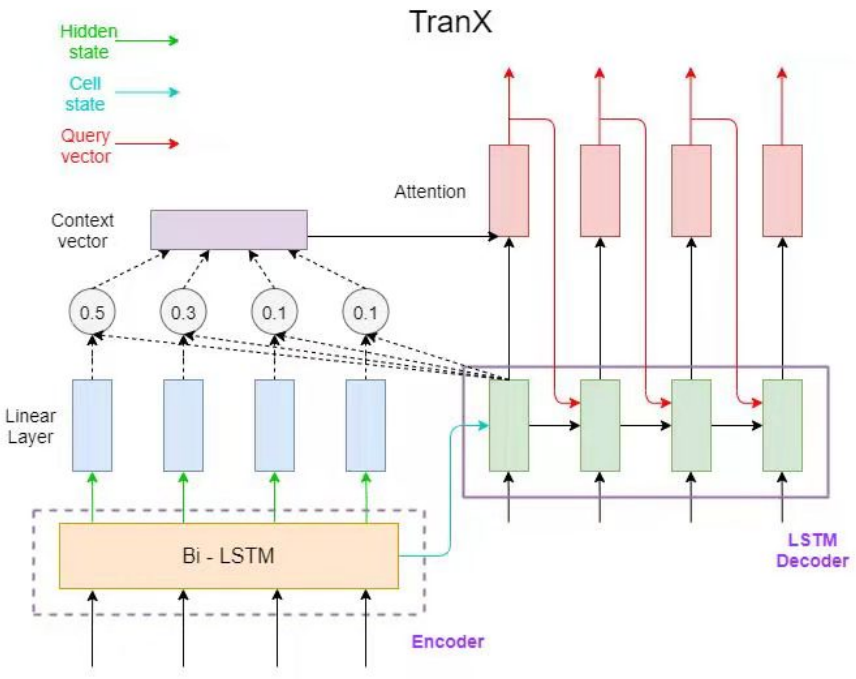

Figure 3: The partial structure of TRANX, with encoder shown in purple dotted line box and decoder shown in purple solid line box

purple dotted line box in Figure 4.

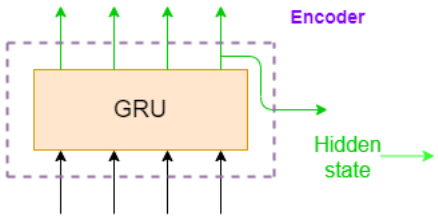

Figure 4: GRU encoder, which corresponding to the change in the purple dashed part of the TRANX partial stucture.

3.2.3 Attentional Encoder

Transformer is an encoder-decoder structure firstly proposed by Vaswani et al. (2017). The recurrent structure of RNN type of models has limitations in the parallelism abilities and information can be easily lost in the process of sequential calculations. Although the structure of gate mechanisms such as LSTM alleviates the problem of long-term dependency to a certain extent, it is still powerless for further-term dependencies. To solve this, Transformer introduces a pure attention-based layer(Vaswani et al., 2017), which is more suitable for exploiting long-term dependencies. In our second attempt, we replaced the LSTM encoder with an attentional encoder. For the sake of fair comparison, we uniformed the input word embedding size to the hidden layer size by adding a linear layer between the input sequence embedding and the attentional encoder.

The default transformer encoder uses 6 layers each with 8 heads, but such configuration is particularly data-hungry and difficult to train. Instead, we use a simplified construct with only 1 layer and 2 heads to merely serve as a proof of concept.

The last vector outputted by the attentional encoder layer corresponds to an $< eos >$ token of the input sequence which contains information of the entire sentence and is used as the initializer of the decoder. The modified part is shown in Figure 5.

3.3 Decoder

As stated above, a standard Bi-LSTM decoder is employed in the architecture. The decoder decodes step-by-step with parent feeding of query vectors. Instead of directly passing in target action embedding as the input of the decoder during training, we concatenate embedding vectors of the previous action and embedding that contain the current frontier information (e.g. ASDL field embedding, ASDL type embedding) to the decoder. After getting the probabilities of each subsequent action, a

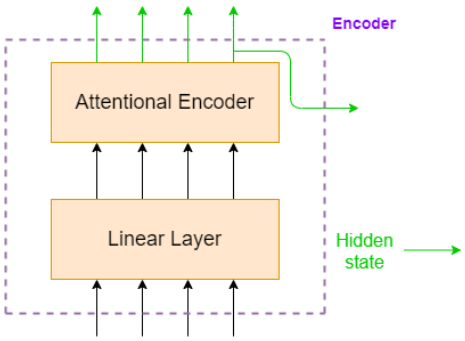

Figure 5: Additional encoder, which corresponding to the change in the purple dashed part of the TRANX partial stucture.

	Example
Question Id	36875258
Intent	Copying one file's contents to another in Python
Rewrite Intent	Copy the content of file 'file.txt' to the file 'file2.txt'.
Code snippet	shutil.copy('file.txt','file2.txt')

Table 1: Example in Manually Curated CoNaLa Dataset

beam search of size 15 is conducted to find the best sequence of actions.

4 Experiments

4.1 Datasets

The CoNaLa dataset released by Carnegie Mellon University (CMU) was used in this project (Yin et al., 2018). This dataset contains mined corpora of natural language intents and Python source code snippets from Stack Overflow. The data is in JSON format and comes in 3 parts: (1) 600k automatically-mined examples, (2) 2379 manually-curated training examples and (3) 500 manually-curated test examples. One manually-curated example is shown in Table 1. Note that the main difference between manually-curated example and automatically-mined example is the missing of 'rewritten_intent' field in mined example.

During our training, we further hold out 200 examples from the training set to serve as a validation set, and the final model performance is reported using the test set.

4.2 Implementation Details

4.2.1 Network

The network structures are as presented in Section 3. All models are initialized using Glorot initialization with Adam as the optimizer. The learning rate is initialized to be 0.001 which decays by a factor of 0.5 every time the $max\_patience$ iterations are hit with no loss improvement. Due to the low inductive bias of the attentional encoder, patience of 10 is set for the third network whereas patience = 5 is set for the LSTM and GRU encoder. The batch size is set as 64 and the dropout ratio is set at 0.3. The details of dimension configurations are presented in Table2.

Vector Name	Dimensions
Word Embedding	128
Action Embedding	128
Field Embedding	64
Type Embedding	64
Hidden Size	256
Query Vector	256

Table 2: Dimension Configurations

It is important to note that the input of the decoder is a concatenation of several vectors: (1) Embedding of the previous action; (2) Embedding of the current action; (3) Embedding of the parent's field; (4) Embedding of the parent's field type; (5) Previous hidden state and (6) Previous query vector.

4.2.2 Project Flow

As mentioned above, the original data is stored in 3 JSON files containing 2879 manually-curated examples and 600k automatically-mined examples. This project uses all of the human-curated data, but only 100k automatically-mined examples for training. After data pre-processing, the NL intents were transformed into lists of tokens, while the Python source code snippets become series of actions through the AST parsing process explained in Section 3.1. One example of the pre-processed data is illustrated in Table 3.

	Example
Source	['Convert', 'a', 'list', 'of', 'integers', 'into', 'a', 'single', 'integer']
Target	r = int(''.join(map(str,x)))

Table 3: Illustration of the Pre-processed Data

We held out 200 examples from the 2379 manually-curated gold training data as the validation set. We first trained the model from scratch with the 100k un-curated examples along with the 200 validation examples. Next, we fine-tuned the obtained model using the 2179 remaining training data and evaluated the same 200 evaluation exam-

	Corpus BLEU	Exact Match	Training Time/Epoch
TRANX(with LSTM)	0.301	0.017	570s
TRANX_GRU	0.286	0.030	400s
TRANX_Attentional_Encoder	0.282	0.024	375s

Table 4: Result Metrics For Candidate Models

ples to form the final model. In the end, we applied the fine-tuned model on the 500 test set on which the results are reported.

4.2.3 Environment

All experiments are conducted on P100 GPUs in Colab. The project also works on the local machine. To run it without CUDA, please simply remove the $--cuda$ flag from all shell scripts under the folder $scripts$.

4.3 Results Evaluation

As shown in Table 4, models with different encoders have competence in different aspects. The original TRANX model outperforms others in terms of the Corpus BLEU by about 0.02, whereas the other two models have less training time per epoch while also achieving comparable performances. In particular, GRU achieves the highest exact match rate whereas the attentional Encoder has the shortest per-epoch training time among the three.

It should be pointed out that though models using GRU and LSTM encoders have similar predicted output, the GRU construct is less inclined to use implicit information. For instance, given the same intent "split string abc", while the model with LSTM encoder predicts "re.split('abc')", the GRU encoder model predicts "abc.split('abc')". This may result from the fact that GRU does not have the cell state to store hidden information. This could be an advantage because rather than understanding the intent, in many cases, the LSTM encoder model selects packages based on keywords, frequently leading to improper use of packages. However, excessive dependence on explicit information also makes it less sensitive to the real meaning of the intent. For example, the GRU model is less sensitive to certain tokens such as "join" and "re". Because of the absence of the cell state, it inherits less information from the last time step, reducing the effect of some common words appearing at the start of the intent text such as "and" and "string" (where "and" often leads to "join" and "string" often leads to "re"). The last point that should be highlighted is that as GRU simplifies the cell structure, it is faster to train than its LSTM sibling despite their similar performances.

As for the Attentional Encoder, it is worthwhile mentioning that it performs well in handling long requirements. It is inclined to generate long codes covering more requirements in the intent. With the help of the attentional layer, long string requirements are separated into small segments and learnt according to their surrounding context. Hence, in this aspect, it outperforms the other two encoders. For instance, given a task that "decode a hex string '4a4b4c' to UTF-8", TRANX generates '4a4b4c . decode (‘ 4a4b4c ‘)', while the attentional encoder gives out a long informative result: 'print (open (‘ 4a4b4c ‘ , ‘ rb ‘) . decode (‘ utf - 8 ‘))'. Additionally, this model deals with words with ambiguous meanings well. For example, when the word 'send' appears in the intent, the other two models' outputs will tend to generate code containing the python function 'send()', while 'send' here is actually an intention. The attentional encoder can distinguish the correct meaning of the words according to its surrounding context by using attention and determine how to interpret the ambiguous words with code. Another advantage of using attentional encoder is that its training time per epoch is the least, indicating its high efficiency. However, this encoder is sensitive to hyperparameters. Improper usage of hyperparameters may lead to failing to converge to the optimal model.

5 Conclusion and Future Work

In this project, we presented the TRANX_GRU and TRANX_attentional_encoder. Both of them are a modification of the current state-of-the-art code generation model TRANX. In particular, TRANX_GRU replaced the LSTM encoder in TRANX with the GRU cells. TRANX_attentional_encoder replaced the recurrent layers used in TRANX architectures with multi-headed self-attention.

For the code generation task based on the CoNaLa dataset, both TRANX_GRU and TRANX_attentional_encoder achieved slightly better results than TRANX in terms of the exact match. In addition, the training time per epoch for our candidate models is less if compared with TRANX. Furthermore, we found that GRU is less inclined to using implicit information. Most predictions are made based on the explicit information showed

directly in the intent. The attentional encoder has a stronger ability in dealing with long context intent and distinguishing ambiguous words, showing a better performance while dealing with complex requirements.

Constrained by time and computation resources, in this project, as a proof of concept, we are only reporting results using the attentional encoder with 1 layer and 2 heads. With such a simple construct, the modified model is already achieving comparable results as TRANX. We also tried to substitute in a complete Transformer model by also replacing the decoder. But since we need to keep all candidate hypotheses during the decoding step, the decoding speed has become the main bottleneck. Also, current configuration of the attentional encoder model is not yet well-tuned. By tuning properly, the performance of the attentional encoder model may even surpass TRANX.

In the future, we will continue the parameter tuning and finding more efficient implementations for the decoding step. Also, instead of using an AST, we will explore the possibilities of generating syntactic structures by applying BERT with proper masks. BERT is a multi-layer bidirectional transformer model which is pre-trained with two tasks: masked language modelling and next sentence predictions (Devlin et al., 2018). It can process sentences and extracts features with multiple focus points. With proper masks, it can also parse tokens in a non-sequential order which resonates with the code generation task where syntactic structures are usually not presented in a sequential order. Via these attempts, we would like to see whether such Transformer based approaches could improve the current TRANX performances by providing better understandings of the long-term dependencies presented in the intent.

The code used to train and evaluate our models is available at https://github.com/kzCassie/ucl_nlp.

References

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

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*.

Susan L Graham. 1980. Table-driven code generation. *Computer*, 13(08):25–34.

Uday Kusupati and Venkata Ravi Teja Ailavarapu. Natural language to code using transformers.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 574–584, Lincoln, Nebraska, USA. IEEE Computer Society.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*.

Muhammad Usman and Aamer Nadeem. 2009. Automatic generation of java code from uml diagrams using ujector. *International Journal of Software Engineering and Its Applications*, 3(2):21–37.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

RJ Waldinger and RCT PROW LEE. 1969. A step toward automatic program writing. In *Proc. Int. Joint Conf. on Artificial Intelligence (Washington DC, May 1969)*, pages 241–252.

Daniel C Wang, Andrew W Appel, Jeffrey L Korn, and Christopher S Serra. 1997. The zephyr abstract syntax description language. In *DSL*, volume 97, pages 17–17.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1693–1703. International World Wide Web Conferences Steering Committee.

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. *arXiv preprint arXiv:1704.01696*.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*.