# TreeGen: A Tree-Based Transformer Architecture for Code Generation

**Zeyu Sun,[†] Qihao Zhu,[†] Yingfei Xiong,[*†] Yican Sun,[†] Lili Mou,[‡] Lu Zhang[†]**
[†]Key Laboratory of High Confidence Software Technologies (Peking University), MoE;
Software Institute, Peking University, 100871, P. R. China
{szy_, zhuqh, xiongyf, sycpku, zhanglucs}@pku.edu.cn
[‡]University of Alberta, Edmonton, AB, Canada
doublepower.mou@gmail.com

## Abstract

A code generation system generates programming language code based on an input natural language description. State-of-the-art approaches rely on neural networks for code generation. However, these code generators suffer from two problems. One is the long dependency problem, where a code element often depends on another far-away code element. A variable reference, for example, depends on its definition, which may appear quite a few lines before. The other problem is structure modeling, as programs contain rich structural information. In this paper, we propose a novel tree-based neural architecture, TreeGen, for code generation. TreeGen uses the attention mechanism of Transformers to alleviate the long-dependency problem, and introduces a novel AST reader (encoder) to incorporate grammar rules and AST structures into the network. We evaluated TreeGen on a Python benchmark, HearthStone, and two semantic parsing benchmarks, ATIS and GEO. TreeGen outperformed the previous state-of-the-art approach by 4.5 percentage points on HearthStone, and achieved the best accuracy among neural network-based approaches on ATIS (89.1%) and GEO (89.6%). We also conducted an ablation test to better understand each component of our model.

## Introduction

Code generation is an important artificial intelligence problem that has the potential to significantly boost the productivity of programmers. Given a specification written in natural language, a code generation system translates the specification into an executable program. For example, if a python programmer gives an instruction "initialize a dictionary, Dict", the code generator is expected to automatically generates "`Dict={ }`".

With the development deep learning techniques, researchers have applied various neural architectures to this problem, such as sequence-to-sequence (Seq2Seq) models or sequence-to-tree (Seq2Tree) models (Sutskever, Vinyals, and Le 2014; Ling et al. 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Hayati et al. 2018; Sun et al. 2019). Especially, state-of-the-art approaches generate code by predicting a sequence of grammar rules (Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Sun et al. 2019). That is to say, the system keeps a partial abstract syntax tree (AST) of the already-generated code, and predicts the grammar rule to be used to expand a particular node.

The classification of grammar rules faces two main challenges. The first challenge is the long-dependency problem (Bengio, Simard, and Frasconi 1994). A code element may depend on another far-away element. For example, a variable reference statement "if $\mathrm{len(a)} < \mathrm{Max\_Length}$:" at line 100 may depend on a variable definition statement "$\mathrm{Max\_Length} = 100$" at line 10. The second challenge is the representation of code structures. It is pointed out that the tree-structural information is crucial for modeling code (Mou et al. 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Sun et al. 2019). However, a "flat" neural architecture, such as an RNN, cannot capture structure information well.

In this paper, we propose a novel neural architecture, TreeGen, for the code generation. To address the first challenge, TreeGen adopts the recently proposed Transformer architecture (Vaswani et al. 2017), which is capable of capturing long dependencies. However, the original Transformer architecture is not designed for programs, and cannot utilize tree structures, i.e., the second above mentioned challenge. A standard way of utilizing structural information, as in graph- and tree-based convolutional neural networks, is to combine the vector representations of a node and its structural neighbors as the output of a structural convolution sub-layer. However, a standard Transformer architecture does not have such structural convolution sub-layers, and it is not clear where to add them.

It is tempting to add structural convolution sub-layers in all the Transformer blocks. Our core conjecture is that when convolving a node and its structural neighbors, the vector representation should mainly contain the information from the original node. As the vector representation of the nodes is processed by more blocks in the decoder of the Transformer, they gradually mix in more information from other nodes and lose their original information. Therefore, we add

[*]Yingfei Xiong is the corresponding author. The code is available at https://github.com/zysszy/TreeGen

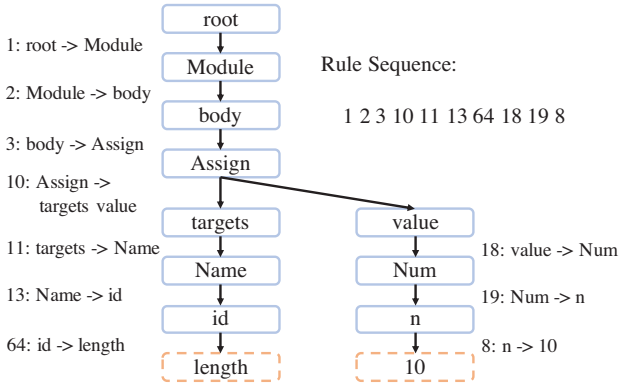


Figure 1: A Python AST for code "$\mathrm{length} = 10$"

the structural convolution sub-layer only to the first several Transformer decoder blocks but not all.

Generally speaking, the TreeGen architecture consists of three parts: (1) a natural language (NL) reader (encoder) encodes the text description; (2) an AST reader (the first several Transformer decoder blocks) encodes the previously generated partial code with the structural convolution sub-layers; (3) a decoder (the rest Transformer decoder blocks) combines the query (the node to be expanded in AST) and the previous two encoders to predict the next grammar rule.

We evaluated our model on an established benchmark dataset for Python code generation, HearthStone (Ling et al. 2016), which is a Python implementation of a card game HearthStone. The results show that our model significantly outperforms previous models by $4.5$ percentage points. We further evaluated our model on two semantic parsing datasets, ATIS and GEO, which translate natural language sentences into lambda calculus logical forms. The results show that our model has the best accuracy among previous neural models, with $89.1\%$ and $89.6\%$ accuracy, respectively. Our evaluation also shows that adding the structural convolution sub-layer to the first several Transformer blocks significantly outperforms a Transformer with structural convolution in all blocks.

## Our Model

We generate code by predicting the grammar rules of the programming language. Figure 2 shows the overall picture of our model, which comprises three parts: an NL reader, an AST reader, and decoder. We introduce them in detail in the following subsections.

### Grammar Rule Prediction

In this section, we introduce how to model code generation as a series of classification problems of grammar rules. The programs can be decomposed into several context-free grammar rules and parsed as an AST. For example, Figure 1 shows a Python AST for the code "$\mathrm{length} = 10$", where dotted boxes are terminal symbols and solid boxes are non-terminal symbols.

AST-based code generation could be thought of as expanding a non-terminal node by a grammar rule. This process is repeated until all leaf nodes are terminal. In Figure 1, "*1: root -> Module*" is an example of the grammar rules, where the preceding number is the ID of the rules. Following the pre-order traverse, we could obtain the sequence of rules that generate the AST shown in the top right corner.

Formally, the probability can be factorized as the probabilities of the rules generating the code following the order.

$$p(\text{code}) = \prod_{i=1}^{P} p(r_i \mid \text{NL input}, r_i, \cdots, r_{i-1}) \qquad (1)$$

where $r_i$ is the $i$th rule in the rule sequence. In this way, our task is to train a model to calculate $p(r_i \mid \text{NL input}, p_i)$, i.e., given the natural language description and the currently generated partial AST the model calculates the probabilities of the rules to expand this node.

### NL Reader

The input description determines the functionality of the code. It can be a semi-structural description as in the HearthStone dataset, or a natural language as in ATIS and GEO semantic parsing datasets.

For an input description, we first tokenize it into tokens $n_1, n_2, \cdots, n_L$, where $L$ denotes the length of the input. Each token $n_i$ is then split to characters $c_1^{(n_i)}, c_2^{(n_i)}, \cdots, c_S^{(n_i)}$, where $S$ is the number of characters in $n_i$. All the tokens and characters are represented as real-valued vectors $\boldsymbol{n}_1, \boldsymbol{n}_2, \cdots, \boldsymbol{n}_L$ and $\boldsymbol{c}_1^{(n_i)}, \boldsymbol{c}_2^{(n_i)}, \cdots, \boldsymbol{c}_S^{(n_i)}$ by *embeddings*.

#### Input Text Representation

*Character Embedding.* It often happens that similar words have similar characters (e.g., "program" and "programs"). To utilize this property, we represent a token by character embeddings with a fully-connected layer

$$\boldsymbol{n}_i^{(\mathrm{c})} = W^{(\mathrm{c})}[\boldsymbol{c}_1^{(n_i)}; \cdots; \boldsymbol{c}_M^{(n_i)}] \qquad (2)$$

where $W^{(\mathrm{c})}$ are the weights and the character sequence is padded to a pre-defined maximum length $M$. After the fully-connected layer, we also apply layer normalization (Lei Ba, Kiros, and Hinton 2016). These vectors are then fed to the NL reader, and are integrated with the word embeddings by a gating sub-layer.

**Neural Structure of NL Reader.** The NL reader is composed of a stack of blocks ($N_d$ blocks in total). Each block contains three different sub-layers (namely, self-attention, gating mechanism, and word convolution) to extract features, which we introduce in detail in the following subsections. Between two sub-layers, we employ a residual connection (He et al. 2016) followed by a layer normalization.

*Self-Attention.* The self-attention sub-layer follows the Transformer's architecture (Vaswani et al. 2017), and uses multi-head attention to capture long dependency information.

For a sequence of input tokens $n_1, n_2, \cdots, n_L$, we represent them as an embedding $\boldsymbol{n}_1, \boldsymbol{n}_2, \cdots, \boldsymbol{n}_L$ by a look-up

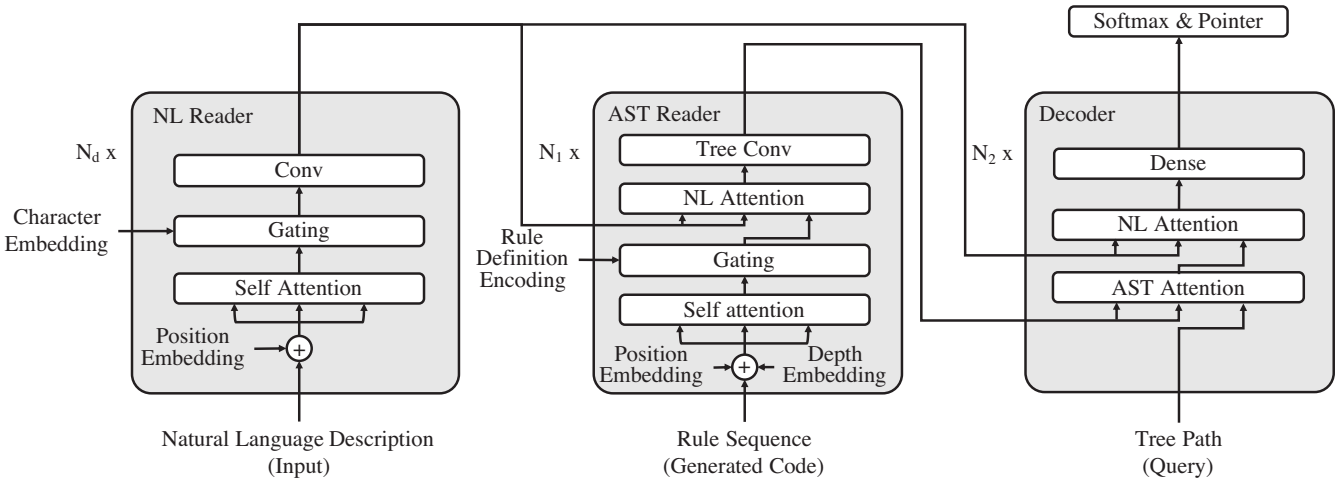


Figure 2: Overview of the TreeGen.

table. We also use position embeddings to encode the information of word positions. In particular, we adopt the variant in Dehghani et al. (2018), and compute the position embedding for the $i$th word in the $b$th Transformer block as

$$p_{b,i}[2j] = \sin((i+b)/(10000^{2j/d})) \tag{3}$$

$$p_{b,i}[2j+1] = \sin((i+b)/(10000^{2j/d})) \tag{4}$$

where $p_{i,b}[\cdot]$ indexes a dimension of the vector $\boldsymbol{p}_{i,b}$, and $d$ is the number of dimensions (i.e., embedding size).

A Transformer block learns non-linear features by multi-head attention, which yields a matrix $Y_b^{(\text{self})} = [\boldsymbol{y}_{b,1}^{(\text{self})}, \boldsymbol{y}_{b,2}^{(\text{self})}, \cdots, \boldsymbol{y}_{b,L}^{(\text{self})}]^\top$, where $Y_b^{(\text{self})} \in \mathbb{R}^{L\times d}$. For notational simplicity, we omit the subscript $b$. The multi-head layer is computed by

$$Y^{(\text{self})} = \text{concat}(head_1, \cdots, head_H)W_h \tag{5}$$

where $H$ denotes the number of heads and $W_h$ is the weight. An attention layer is applied in each head $head_t$, computed by

$$head_t = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V \tag{6}$$

where $d_k = d/H$ denotes the length of each features vector. $Q$, $K$ and $V$ are computed by

$$[Q, K, V] = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_L]^\top [W_Q, W_K, W_V] \tag{7}$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d\times d_k}$ are model parameters. $\boldsymbol{x}_i$ is the input of this Transformer block. For the first block, it is the vector sum of the table-lookup embedding and the position embedding, i.e., $\boldsymbol{n}_i + \boldsymbol{p}_{1,i}$; for other blocks, it is the vector sum of the lower Transformer block's output and the position embedding that corresponds to this block.

*Gating Mechanism.* After the features are computed by self-attention, we further incorporate with the information of character embeddings. This is given by a gating mechanism based on softmax. For the $i$th word, we compute a control vector $\boldsymbol{q}_i$ from $\boldsymbol{y}_i^{(\text{self})}$ by a linear transformation. The softmax weight $\boldsymbol{k}_i^{(\text{c})}$ for character embedding is given by a linear transformation from $\boldsymbol{n}_i^{(\text{c})}$ in Equation 2. The softmax weight $\boldsymbol{k}_i^{(\text{y})}$ for Transformer's output is given by another linear transformation from $\boldsymbol{y}_i^{(\text{self})}$. Then, the gate is computed by

$$[\alpha_{i,t}^{(\text{y})}, \alpha_{i,t}^{(\text{c})}] = \text{softmax}\{\boldsymbol{q}_i^\top \boldsymbol{k}_i^{(\text{y})}, \boldsymbol{q}_i^\top \boldsymbol{k}_i^{(\text{c})}\} \tag{8}$$

They are used to weigh the feature of the Transformer's layer $\boldsymbol{v}_i^{(\text{y})}$ and the feature of character embeddings $\boldsymbol{v}_i^{(\text{c})}$, linear transformed from $\boldsymbol{y}_i^{(\text{self})}$ and $\boldsymbol{n}_i^{(\text{c})}$, respectively.

$$\boldsymbol{h}_{i,t} = [\alpha_{i,t}^{(\text{y})} \boldsymbol{v}_i^{(\text{y})} + \alpha_{i,t}^{(\text{c})} \boldsymbol{v}_i^{(\text{c})}] \tag{9}$$

Similar to Equation 5, the output of our gating mechanism is $Y^{(\text{gate})} = (\boldsymbol{h}_{i,t})_{i,t}$, where $(\cdot)_{i,t}$ represents a block matrix with the elements being $h_{i,t}$.

*Word Convolution.* Finally, two convolutional layers are applied to the output of the gating mechanism $\boldsymbol{y}_1^{(\text{gate})}, \cdots, \boldsymbol{y}_L^{(\text{gate})}$ and to extract the local features around each token $\boldsymbol{y}_1^{(\text{conv},l)}, \cdots, \boldsymbol{y}_L^{(\text{conv},l)}$, where $l$ denotes the layer of convolutional layers. The $\boldsymbol{y}_i^{(\text{conv},l)}$ is computed by

$$\boldsymbol{y}_i^{(\text{conv},l)} = W^{(\text{conv},l)}[\boldsymbol{y}_{i-w}^{(\text{conv},l-1)}; \cdots; \boldsymbol{y}_{i+w}^{(\text{conv},l-1)}] \tag{10}$$

where $W^{(\text{conv},l)}$ are the convolution weights, $w = (k-1)/2$, and $k$ denotes the window size. In particular, $\boldsymbol{y}_i^{(\text{conv},0)}$ denotes the output of gating mechanism $\boldsymbol{y}_i^{(\text{gate})}$. In these layers, separable convolution (Chollet 2017) is used. The reason is separable convolution has fewer parameters that is easy for training. For the first and the last words, we add zero padding. Between these layers, we use the GELU activation function (Hendrycks and Gimpel 2016).

In summary, the NL reader has a few Transformer blocks of self-attention, the gating mechanism, and word convolution. The natural language description is encoded as features $\boldsymbol{y}_1^{(\text{NL})}, \boldsymbol{y}_2^{(\text{NL})}, \cdots, \boldsymbol{y}_L^{(\text{NL})}$.

## AST Reader

We design an AST reader to model the structure of the partial AST that has generated. Although our programs are generated by predicting the sequence of grammar rules, these rules alone lack a concrete picture of the program and are insufficient for predicting the next rule. Therefore, our AST reader considers heterogeneous information, including the predicted rules and the tree structures.

To incorporate such program-specific information, we first represent the code as a sequence of rules, then encode the rules with attention mechanism, and finally use a tree convolution layer to combine the encoded representation of each node with its ancestors.

### AST Representation

*Rule Sequence Embedding.* To encode the rule information, we use the ID of the rules. Suppose we have a sequence of rules $r_1, r_2, \cdots, r_P$ that are been used to generate the partial AST in a decoding step, where $P$ denotes the length of the sequence. We represent these rules as real-valued vectors $\boldsymbol{r}_1, \boldsymbol{r}_2, \cdots, \boldsymbol{r}_P$ by table-lookup embeddings.

*Rule Definition Encoding.* The above table-lookup embedding treats a grammar rule as an atomic token, and loses the information of the rule's content.

To alleviate this problem, we enhance the representation of a rule with the encoding of rule definition.

For a grammar rule $i : \alpha \rightarrow \beta_1 \cdots \beta_K$, where $\alpha$ is the parent node and $\beta_1 \cdots \beta_K$ are child nodes. They can be either terminal or non-terminal symbols. The index $i$ is the ID of the rule.

Similar to Equation 2, we encode the rule content as a vector $\boldsymbol{r}^{(\text{c})}$ by a fully-connected layer with input being the table-lookup embeddings $\boldsymbol{\alpha}, \boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ of respective symbols. It is noted that the sequence is also padded to a maximum length.

Then the rule definition features $\boldsymbol{y}_1^{(\text{rule})}, \cdots, \boldsymbol{y}_P^{(\text{rule})}$ are computed by another fully-connected layer as

$$\boldsymbol{y}_i^{(\text{rule})} = W^{(\text{rule})}[\boldsymbol{r}_i; \boldsymbol{r}^{(\text{c})}; \boldsymbol{\alpha}] \tag{11}$$

where $\boldsymbol{r}_i$ is the table-lookup embedding of the rule $r_i$, $\boldsymbol{r}_i^{(\text{c})}$ is the content-encoding rule representation, and we emphasize the parent node $\alpha$ again. After that, a layer normalization is followed.

*Position and Depth Embeddings.* Since our AST reader would use self-attention mechanisms, we need to represent the position where a grammar rule is used.

We first adopt the position embedding as in Equation 4, representing when a rule is used in the sequence $r_1, \cdots, r_P$. The position embeddings are denoted by $\boldsymbol{p}_1^{(\text{r})} \cdots, \boldsymbol{p}_P^{(\text{r})}$

However, such position embedding does not capture the position of a rule in the AST. We further encode such information by a *depth embedding*. If we expand a symbol $\alpha$ by the rule $r : \alpha \rightarrow \beta_1 \cdots \beta_K$, we represent the depth of the rule by its parent node, i.e., $\alpha$. In this way, we associate another sequence of table-lookup depth embeddings $\boldsymbol{d}_1, \cdots, \boldsymbol{d}_P$ to the sequence of used grammar rules $\boldsymbol{r}_1, \cdots, \boldsymbol{r}_P$.

**Neural Structure of AST Reader.** The AST reader is also composed of a stack of blocks ($N_1$ blocks in total). Each block is decomposed into four sub-layers (namely, self-attention, a gating mechanism, NL attention, and tree convolution). We employ a residual connection around each sub-layer except the layer of tree convolution. After each sub-layer, we apply a layer normalization.

*Self-Attention.* To capture the information of AST, we build a Transformer-like self-attention layer, where the input is sum of the rule embedding, position embedding, and depth embedding, i.e., $\boldsymbol{r}_i + \boldsymbol{d}_i + \boldsymbol{p}_i^{(\text{r})}$. The self-attention sub-layer extract features $\boldsymbol{y}_1^{(\text{ast-self})}, \boldsymbol{y}_2^{(\text{ast-self})}, \cdots, \boldsymbol{y}_P^{(\text{ast-self})}$ of AST input, using the same mechanism as Equations 4, 5, 6 with different weights but add an additional depth embedding to $\boldsymbol{p}_i^{(\text{r})}$.

*Gating Mechanism.* We would like to incorporate the content-encoding rule representation $\boldsymbol{y}_i^{(\text{rule})}$ into the Transformer-extracted features. We adopt a gating mechanism as in Equations 8, 9, and the fused features becomes $\boldsymbol{y}_1^{(\text{ast-g})}, \boldsymbol{y}_2^{(\text{ast-g})}, \cdots, \boldsymbol{y}_P^{(\text{ast-g})}$ after this sub-layer.

*NL Attention.* During the decoding step, we should be informed of the input NL description. This is given by a multi-head NL attention, similar to the Transformer decoder's attention to its encoder (Vaswani et al. 2017). The extracted features are denoted by$\boldsymbol{y}_1^{(\text{ast-nl})}, \boldsymbol{y}_2^{(\text{ast-nl})}, \cdots, \boldsymbol{y}_P^{(\text{ast-nl})}$.

*Tree Convolution.* Should we consider only the above sub-layers, it would be hard for the reader to combine the information of a node with its ancestors. A node can be far away from its ancestors in the rule sequence but is close in structure. Therefore, it is difficult for a traditional Transformer to extract such structural features.

We integrate the features of a node with those of its ancestors. We treat the AST as a graph and use an adjacency matrix $M$ to represent the directed graph. If a node $\alpha_i$ is the parent of $\alpha_j$, then $M_{ji} = 1$. Suppose all the nodes are presented by features $\boldsymbol{f}_1, \cdots, \boldsymbol{f}_n$, their parents' features can be given by the multiplication with the adjacency matrix:

$$[\boldsymbol{f}_1^{(\text{par})}, \cdots, \boldsymbol{f}_n^{(\text{par})}] = [\boldsymbol{f}_1, \cdots, \boldsymbol{f}_n]M \tag{12}$$

where $\boldsymbol{f}_i^{(\text{par})}$ denotes the parent of the $i$th node. For the father of the root node, we pad it with the feature vector of the root node itself.

The tree-based convolution window, applied to the current sub-tree, is given by

$$\begin{aligned} Y^{(\text{tconv},\, l)} = f(W^{(\text{tconv},\, l)}[&Y^{(\text{tconv},\, l-1)}; \\ &Y^{(\text{tconv},\, l-1)}M; \cdots; Y^{(\text{tconv},\, l-1)}M^{kt-1}]) \end{aligned} \tag{13}$$

where $W^{(\text{tconv},\, l)}$ is the weights of the convolutional layer, $kt$ denotes the window size (we set to 3 in our experiments), $l$ is the layer of these convolutional layers. In particular, $Y^{(\text{tconv},\, 0)} = [\boldsymbol{y}_1^{(\text{att})}, \boldsymbol{y}_2^{(\text{att})}, \cdots, \boldsymbol{y}_P^{(\text{att})}]$, where $Y^{(\text{tconv},\, 0)} \in \mathbb{R}^{d \times P}$. For the last layer of the AST reader, we add additional two convoluation layers. In the equation, $f$ is the activation function and GELU is applied between these layers.

In summary, the AST reader has $N_1$ blocks of these four sub-layers, and yields the features $\boldsymbol{y}_1^{(\text{ast})}, \boldsymbol{y}_2^{(\text{ast})}, \cdots, \boldsymbol{y}_P^{(\text{ast})}$.

## Decoder

Our final component is a decoder that integrates the information of the generated code with the NL description, and predicts the next grammar rule. Similar to the AST reader, a stack of blocks ($N_2$ blocks in total) each with several sub-layers is used in the decoder as follows. A residual connection is also employed around each sub-layer followed by a layer normalization.

The decoder takes the non-terminal node to be expanded as a query. Inspired by a previous approach (Sun et al. 2019), the querying node is represented as a path from the root to the node to be expanded. For example, if we are going to expand node "Assign" in Figure 1, the path should be *root, Module, body, Assign*. We represent the nodes in this path as real-valued vectors. Then we apply a fully-connected layer like Equation 2 to these vectors and the output of the path (querying node) is $\boldsymbol{q}_i^{(\text{path})}$.

We then apply two attention layers to integrate the outputs of the AST reader and the NL reader.

We first apply an AST attention layer over the output of the AST reader with queries and extract features $\boldsymbol{f}_1^{(\text{tree})}, \cdots, \boldsymbol{f}_P^{(\text{tree})}$. In this layer, $Q$ is computed from queries $\boldsymbol{q}_1^{(\text{path})}, \cdots, \boldsymbol{q}_P^{(\text{path})}$; $K$ and $V$ are computed from the code features $\boldsymbol{y}_1^{(\text{ast})}, \cdots, \boldsymbol{y}_P^{(\text{ast})}$. We further integrate the features from the input description. This integration is also implemented with an NL attention, where $Q$ is computed by feature $\boldsymbol{f}_1^{(\text{tree})}, \cdots, \boldsymbol{f}_P^{(\text{tree})}$; and $K$ and $V$ are computed by the input description $\boldsymbol{y}_1^{(\text{NL})}, \cdots, \boldsymbol{y}_L^{(\text{NL})}$.

Finally, a set of two fully-connected layers, where the first layer has a $GELU$ activation function, are followed to extract features for prediction.

## Training and Inference

We predict the next grammar rule, among all possible candidates, by softmax based on the decoder's last layer features.

We also introduce the pointer network (See, Liu, and Manning 2017) (essentially, an attention) that can directly copy a token $a$ from the NL description. In this case, the resulting grammar rule is $\alpha \rightarrow a$, where $\alpha$ is a non-terminal symbol to be expanded and $a$ is a terminal symbol. Such pointer mechanism is helpful for user-defined identifiers (e.g., variable and function names).

The choice between softmax rule prediction and the pointer network is given by another gating mechanism $p_g$, also computed from the decoder's last feature. The overall predicted probability of the next grammar rule is

$$p(r_i|\cdot) = \begin{cases} p_g\, p(r_i|\cdot) & \text{if } i \in \mathbf{D} \\ (1-p_g)\,\Pr\{\text{copy word } t \text{ at step } i|\cdot\} & \text{if } i \in \mathbf{C} \end{cases} \tag{14}$$

where $i$ denotes the ID of the rule, $\mathbf{D}$ is the set of predefined rules, and $\mathbf{C}$ denotes the set of rules in the form of $\alpha \rightarrow a$, where $a$ is a terminal token that occurs in the NL description.

$p_g$ is the probability of using the type of predefined rules, and the $p(r_i|\cdot)$ (the probability of each predefined rules) are computed by two single-layer perceptrons with the sigmoid and softmax activation functions, respectively, and the input of these layers are the features $\boldsymbol{h}^{(\text{dec})}$.


NAME: Darkscale Healer
ATK: 4
DEF: 5
COST: 5
DUR: -1
TYPE: Minion
PLAYER: Neutral
RACE: NIL
RARITY: Common
DESCRIPTION: Battlecry: Restore 2 Health to all friendly characters.

```
class DarkscaleHealer(MinionCard):
    def __init__ (self) :
        super().__init__("Darkscale Healer", 5,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry = Battlecry(Heal(2),
                CharacterSelector()))

    def create_minion (self, player):
        return Minion(4, 5)
```

Figure 3: A example of the implement of HearthStone.

The pointer network is computed by

$$\xi_t = \boldsymbol{v}^T \tanh(W_1 \boldsymbol{h}^{(\text{dec})} + W_2 \boldsymbol{y}_t^{(\text{NL})})$$
$$\Pr\{\text{copy word } t \text{ at step } i|\cdot\} = \frac{\exp\{\xi_t\}}{\sum_{j=1}^{L} \exp\{\xi_j\}} \tag{15}$$

where $\boldsymbol{h}^{(\text{dec})}$ denotes the decoder's last feature. The model is optimized by maximizing negative log likelihood loss against the reference program.

The inference starts with a *start* rule, $start : snode \longrightarrow root$, expanding a special symbol *snode* to the *root* symbol. The recursive prediction terminates if every leaf node in the predicted AST is a terminal. During prediction, we use beam search with a size of 5. Invalid rules are excluded during beam search.

# Evaluation

We evaluated our approach on two types of benchmarks: (1) a Python code generation benchmark, HearthStone, and (2) two semantic parsing benchmarks, ATIS and GEO.

## Experiment: HearthStone

**Dataset.** We first evaluated our approach on the HearthStone benchmark (Ling et al. 2016). The benchmark contains Python code that implements 665 different cards of HearthStone. Each card is composed of a semi-structural description and a groundtruth Python program. The Python programs have a length of 84 tokens on average. The description comes with several attributes such as card name, card type, as well as a natural language description for the functionality of the card. A Python program is mainly decided by the natural language description where the attributes decide the constants or identifier names. A sample description and its corresponding Python program are shown in Figure 3. When preprocessing the card description into token sequences, existing approaches consider two methods.

The first (Yin and Neubig 2017; Hayati et al. 2018) (called *plain* preprocessing) treats the whole description as plain text and delimit the tokens by standard separators such as space or periods. The second (Rabinovich, Stern, and Klein 2017) (called *structural* preprocessing) treats the descriptions as semi-structural and always treat an attribute as one token. In this experiment, we consider both methods and denote the results corresponding to the plain preprocessing as TreeGen-A and that corresponding to the structural preprocessing as TreeGen-B. We followed the train-dev-test split in Ling et al. (2016), and the statistic is listed in Table 2.

**Metrics.** We measured the performance following the metrics in Sun et al. (2019). We computed the StrAcc, which is the percentage of programs that has exactly the same token sequence as the ground truth; the BLEU score, which is used to measure the similarity between the generated code and the reference code at the token level; and the Acc+, which is evaluated manually, allows variable renaming on top of StrAcc, for every test case.

**Settings.** For neural networks, we set the number of NL reader layers $N_d = 6$, and $N_1 = N_2 = 5$ for the AST reader as well as the decoder. The size of all embedding is 256. The hidden sizes were all set to the 256 except each fully-connected layers, except the first layer was 1024 dimensions. We applied dropout after each layer (including attention layers, gating mechanism layers, convolutional layers, and fully-connected layers, where the drop rate is 0.15). The model is optimized by Adafactor (Shazeer and Stern 2018) with default parameters.

**Overall Results.** We show the results in Table 1. In this table, the structural preprocessing has a better performance compared with the plain preprocessing.

As shown, our model achieves 6 percentage points accuracy improvement with plain preprocessing and 4.5 percentage points accuracy improvement with structural preprocessing. For the BLEU score, our model also achieves the best results. These boosts in performance indicate that TreeGen successfully alleviates the long dependency problem and effectively encodes the structural information in the code generation.

**Time Efficiency.** We further evaluated the complexity of our model on the HearthStone, and the result shows that our model is faster than the previous ones. It takes 18s for an epoch on a single Nvidia Titan XP, whereas 180s for the CNN (Sun et al. 2019) and 49s for the RNN (Yin and Neubig 2017).

**Location of Structural Convolution Sub-layer.** One of the keys of our approach is to add the structural convolution sub-layers only to part of the Transformer blocks in the decoder. To evaluate whether this design decision is effective, we evaluate four competing settings: 1) adding the structural convolution sub-layers to all Transformer blocks (i.e., $N_1 = 10$); 2) adding the structural convolution sub-layers to the first 7 blocks in AST reader (i.e., $N_1 = 10(7)$); 3) adding the structural convolution sub-layers to the first 8 blocks in AST reader (i.e., $N_1 = 10(8)$); 4) the other adds to none (i.e., $N_1 = 0$). As we can see, from Table 1 our approach adding the sub-layer to all transformer blocks ($N_1 = 10$) significantly outperforms the last setting ($N_1 = 0$), but slightly worse than the other two settings.

**Ablation Test.** We ablated our model (TreeGen-B was used) to analyze the contribution of each component, results also shown in Table 1. First, we compared our model with the traditional Transformer, which is a Transformer without effective structure modeling. We achieved 21 percentage points higher accuracy (p-value is less than 0.001) and 12 higher BLEU score. This result provides strong evidence of the effectiveness of the AST reader in our model and the importance of the structural information. Next, we replaced the tree convolutional layers in the AST Reader with two layers of fully-connected layers, and we removed the char embedding, rule definition encoding, self-attention layers in turn. The experimental results show the identifiers-encoding, alleviating long-dependency and structural information significantly influence the accuracy. Please note that in some cases BLEU increases while StrAcc and Acc+ decrease. Here we consider StrAcc and Acc+ more important as they guarantee the correctness of the generated programs and correctness is usually crucial in code generation.

## Experiment II: Semantic Parsing

**Dataset.** We further evaluated our approach on the semantic parsing tasks. Our experiment was conducted on two semantic parsing datasets, ATIS and GEO. The input of these datasets is a natural language description, while the output is a short piece of code in lambda calculus. We followed the standard train-dev-test split of these datasets, and the statistics are listed in Table 2.

**Metrics and Settings.** In this task, we follow the evaluation of the previous approaches (Dong and Lapata 2016) and use accuracy as the metric, where the tree exact match was considered to avoid spurious errors. In other words, the order of the children can be changed within conjunction nodes. We followed all the settings in the HS experiment except that we changed the embedding size and the hidden sizes to 128 compared with the setting of the HS experiment.

**Results.** Table 3 shows the performance of our TreeGen. As seen, the accuracy of our approach is sightly worse than the traditional approach WKZ14 (Wang, Kwiatkowski, and Zettlemoyer 2014), which is based on the CCG parser and uses a large number of templates. This traditional approach is hard to generalize new datasets. However, our model was directly adopted from the HS dataset, and achieved the highest accuracy, among all neural models (Dong and Lapata 2016; Rabinovich, Stern, and Klein 2017; Dong and Lapata 2018; Chen, Sun, and Han 2018; Xu et al. 2018;

| | Model | StrAcc | Acc+ | BLEU |
|---|---|---|---|---|
| Plain | LPN (Ling et al. 2016) | 6.1 | – | 67.1 |
| | SEQ2TREE (Dong and Lapata 2016) | 1.5 | – | 53.4 |
| | YN17 (Yin and Neubig 2017) | 16.2 | ∼18.2 | 75.8 |
| | ASN (Rabinovich, Stern, and Klein 2017) | 18.2 | – | 77.6 |
| | ReCode (Hayati et al. 2018) | 19.6 | – | 78.4 |
| | **TreeGen-A** | **25.8** | **25.8** | **79.3** |
| Structural | ASN+SUPATT (Rabinovich, Stern, and Klein 2017) | 22.7 | – | 79.2 |
| | SZM19 (Sun et al. 2019) | 27.3 | 30.3 | 79.6 |
| | **TreeGen-B** | **31.8** | **33.3** | 80.8 |
| | **Location of Structural Convolutional Sub-layer** | | | |
| | $N_1 = 10, N_2 = 0$ | 25.8 | 27.3 | 80.4 |
| | $N_1 = 10(7), N_2 = 0$ | 27.3 | 28.8 | 78.5 |
| | $N_1 = 10(8), N_2 = 0$ | 25.8 | 28.8 | 78.5 |
| | $N_1 = 0, N_2 = 10$ | 21.2 | 22.7 | 79.6 |
| | **Ablation test** | | | |
| | Baseline: Transformer | 10.6 ($p = 0.015$) | 12.1 | 68.0 |
| | - Tree Convolution | 27.3 ($p = 0.015$) | 27.3 | 80.9 |
| | - Rule Definition Encoding | 27.3 ($p < 0.001$) | 28.8 | **81.8** |
| | - Char Embedding | 15.2 ($p < 0.001$) | 18.2 | 72.9 |
| | - Self-Attention | 28.8 ($p < 0.001$) | 28.8 | 81.0 |

Table 1: Performance of our model in comparison with previous state-of-the-art results.

| | | Exp II | |
|---|---|---|---|
| **Statistics** | **HS** | **ATIS** | **GEO** |
| # Train | 533 | 4,434 | 600 |
| # Dev | 66 | 491 | - |
| # Test | 66 | 448 | 280 |
| Avg. tokens in description | 35.0 | 10.6 | 7.4 |
| Max. tokens in description | 76.0 | 48 | 23 |
| Avg. tokens in code | 83.2 | 33.9 | 28.3 |
| Max. tokens in code | 403 | 113 | 144 |

Table 2: Statistics of the datasets we used.

| | Method | ATIS | GEO |
|---|---|---|---|
| Traditional | ZC07 (Zettlemoyer and Collins 2007) | 84.6 | 86.1 |
| | FUBL (Kwiatkowski et al. 2011) | 82.8 | 88.6 |
| | KCAZ13 (Kwiatkowski et al. 2013) | - | 89.0 |
| | WKZ14 (Wang, Kwiatkowski, and Zettlemoyer 2014) | **91.3** | **90.4** |
| Neural Networks | SEQ2SEQ (Dong and Lapata 2016) | 84.2 | 84.6 |
| | SEQ2TREE (Dong and Lapata 2016) | 84.6 | 87.1 |
| | ASN (Rabinovich, Stern, and Klein 2017) | 85.3 | 85.7 |
| | ASN+SUPATT (Rabinovich, Stern, and Klein 2017) | 85.9 | 87.1 |
| | COARSE2FINE (Dong and Lapata 2018) | 87.7 | 88.2 |
| | TRANX (Yin and Neubig 2018) | 86.2 | 88.2 |
| | Seq2Act (Chen, Sun, and Han 2018) | 85.5 | 88.9 |
| | Graph2Seq (Xu et al. 2018) | 85.9 | 88.1 |
| | SZM19 (Sun et al. 2019) | 85.0 | - |
| | **TreeGen** | **89.1** | **89.6** |

Table 3: Accuracy in semantic parsing (in percentage).

Sun et al. 2019). This experiment shows the effectiveness and generalizability of TreeGen.

## Related Work

Code generation achieves significant progress in recent years. The early approaches are mainly based on templates (Zettlemoyer and Collins 2007; 2005; Kushman and Barzilay 2013; Wang, Kwiatkowski, and Zettlemoyer 2014). With the prosperity of deep learning, the sequence-to-sequence framework has shown to be effective in various tasks (Sutskever, Vinyals, and Le 2014). Ling et al. (2016) applied this framework to generate code based on tokens. Unlike natural languages, it is shown that the code contains much more structural information. Thus, the abstract syntax tree (AST) was used in more recent works (Dong and Lapata 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017; Hayati et al. 2018; Yin and Neubig 2018). However, these studies mainly use recurrent neural networks (RNNs) from the long dependency problem (Bengio, Simard, and Frasconi 1994). Sun et al. (2019) proposed to use the convolutional neural network (CNN) to handle the long dependency problem. Our approach addresses this problem by Transformer's intensive attention mechanism (Vaswani et al. 2017). To incorporate the structural information and the idea of self-attention, we propose a tree-based Transformer architecture for code generation.

## Conclusion

In this work, we propose TreeGen for program generation. TreeGen uses the attention mechanism of Transformers to alleviate the long-dependency problem and introduces the AST reader to combine the grammar rules and the AST structure.

The evaluation was conducted on a Python dataset, HearthStone, and two semantic parsing datasets, ATIS and GEO. The experimental results show that our model signifi-

cantly outperforms existing approaches. We also conducted in-depth ablation tests, which suggests that each component in our model plays a significant role.

## Acknowledgments

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, and National Natural Science Foundation of China under Grant Nos. 61672045, 61529201, and 61922003. Lili Mou is an Amii Fellow; he is supported by the CCAI Chair Program; and he also thanks AltaML for support.

## References

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* 5(2):157–166.

Chen, B.; Sun, L.; and Han, X. 2018. Sequence-to-Action: End-to-End Semantic Graph Generation for Semantic Parsing. In *ACL*, 766–777.

Chollet, F. 2017. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, 1251–1258.

Dehghani, M.; Gouws, S.; Vinyals, O.; Uszkoreit, J.; and Kaiser, Ł. 2018. Universal transformers. *arXiv preprint arXiv:1807.03819*.

Dong, L., and Lapata, M. 2016. Language to Logical Form with Neural Attention. In *ACL*, 33–43.

Dong, L., and Lapata, M. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *ACL*, 731–742.

Hayati, S. A.; Olivier, R.; Avvaru, P.; Yin, P.; Tomasic, A.; and Neubig, G. 2018. Retrieval-Based Neural Code Generation. In *EMNLP*, 925–930.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *CVPR*, 770–778.

Hendrycks, D., and Gimpel, K. 2016. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *arXiv preprint arXiv:1606.08415*.

Kushman, N., and Barzilay, R. 2013. Using semantic unification to generate regular expressions from natural language. In *NAACL*, 826–836.

Kwiatkowski, T.; Zettlemoyer, L.; Goldwater, S.; and Steedman, M. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *EMNLP*, 1512–1523.

Kwiatkowski, T.; Choi, E.; Artzi, Y.; and Zettlemoyer, L. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *EMNLP*, 1545–1556.

Lei Ba, J.; Kiros, J. R.; and Hinton, G. E. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Ling, W.; Blunsom, P.; Grefenstette, E.; Hermann, K. M.; Kočiskỳ, T.; Wang, F.; and Senior, A. 2016. Latent Predictor Networks for Code Generation. In *ACL*, 599–609.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 1287–1293.

Rabinovich, M.; Stern, M.; and Klein, D. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*, 1139–1149.

See, A.; Liu, P. J.; and Manning, C. D. 2017. Get to the point: Summarization with pointer-generator networks. In *ACL*, 1073–1083.

Shazeer, N., and Stern, M. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. *arXiv preprint arXiv:1804.04235*.

Sun, Z.; Zhu, Q.; Mou, L.; Xiong, Y.; Li, G.; and Zhang, L. 2019. A grammar-based structural cnn decoder for code generation. In *AAAI*, volume 33, 7055–7062.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*, 3104–3112.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *NIPS*, 6000–6010.

Wang, A.; Kwiatkowski, T.; and Zettlemoyer, L. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *EMNLP*, 1284–1295.

Xu, K.; Wu, L.; Wang, Z.; Yu, M.; Chen, L.; and Sheinin, V. 2018. Exploiting Rich Syntactic Information for Semantic Parsing with Graph-to-Sequence Model. In *ACL*, 918–924.

Yin, P., and Neubig, G. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*, 440–450.

Yin, P., and Neubig, G. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *EMNLP*, 7–12.

Zettlemoyer, L. S., and Collins, M. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *UAI*, 658–666.

Zettlemoyer, L., and Collins, M. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL*, 678–687.