Grape: Grammar-Preserving Rule Embedding

Qihao Zhu , Zeyu Sun , Wenjie Zhang , Yingfei Xiong* , Lu Zhang

Key Laboratory of High Confidence Software Technologies (Peking University), MoE;
Software Institute, Peking University, 100871, P. R. China

{zhuqh, szy_, zhang_wen_jie,xiongyf,zhanglucs}@pku.edu.cn

Abstract

Word embedding has been widely used in various areas to boost the performance of the neural models. However, when processing context-free languages, embedding grammar rules with word embedding loses two types of information. One is the structural relationship between the grammar rules, and the other one is the content information of the rule definition. In this paper, we make the first attempt to learn a grammar-preserving rule embedding. We first introduce a novel graph structure to represent the context-free grammar. Then, we apply a Graph Neural Network (GNN) to extract the structural information and use a gating layer to integrate content information. We conducted experiments on six widely-used benchmarks containing four context-free languages. The results show that our approach improves the accuracy of the base model by 0.8 to 6.4 percentage points. Furthermore, Grape also achieves 1.6 F1 score improvement on the method naming task which shows the generality of our approach.

1 Introduction

Learning a representation for a basic unit (e.g., words in a sentence) plays an increasingly essential role in natural language processing tasks [Wolf *et al.*, 2014; Sienčnik, 2015]. In the form of a real-valued vector, the most widely used word embedding encodes the semantics of a word in a context. Existing approaches mostly focus on how to learn a meaningful representation of words in a natural language (NL) [Mikolov *et al.*, 2013; Pennington *et al.*, 2014].

As we move to a scenario in which embedding techniques routinely represent tokens in context-free languages (e.g., programming languages, domain specific languages, regular expressions), early approaches directly treat the tokens as words in a NL and adopt word embedding techniques. Different from the sentence in NL, the context-free sentence (CFS) [1] can be parsed and further compiled by machines, which highly relies on the pre-defined grammar. For example, if we remove ";" from "int i = 0;", the C++ compiler will report a grammar error and fail to compile the CFS. This highlights the importance of the grammar.

Mou *et al.* [2016] has realized the fact and proposes to embed the grammar information by parsing the CFS as an abstract syntax tree (AST). The AST is further represented as a sequence of production rules by more recent work [Yin and Neubig, 2017; Rabinovich *et al.*, 2017; Yin and Neubig, 2018; Iyer *et al.*, 2018; Sun *et al.*, 2020]. Unfortunately, although they transfer the CFS into the production rule sequence, they still use the word embedding technique to represent these rules, which does not utilize the meta-level information in the grammar definition, e.g., $b \to c$ can be used after $a \to b$ to further expand b.

In this paper, we propose a **Gra**mmar-**P**reserving Rule **E**mbedding approach called Grape. Similar to word embedding, our approach produces the embeddings of the grammar rules that can be used in downstream applications. Unlike word embedding, we preserve the information of the predefined context-free grammar rules.

However, it is challenging to preserve such grammars. First, context-free grammar contains rich structural information, i.e., the structural relationships between different grammar rules, such as whether a rule can be a parent of another rule in an AST. If such information is preserved, structurally similar rules should have similar embeddings. For example in Figure 1, either rule 4 or rule 5 can be a child of rule 3, and also be a parent of rule 6, so their embeddings are better to be close. Second, context-free grammar also contains extensive content information, i.e., the information in the content of the rule definition itself, such as the symbols used in the rule, the order of the symbols, etc. For example, the embedding of rule 6 and rule 9 should be similar as they have the shared symbol *orelse*. Word embedding can preserve neither type of information because the rules are represented using one-hot encoding, such that neither the structural relation of the rules nor the content of the rules are used as input for the neural network.

To preserve the structural information, we propose to use a graph to represent the context-free grammar, called a *grammar relation graph*, where each node in the graph corre-

*Corresponding author.

[1] A text written by the corresponding context-free grammar. In this paper, we only talk about the grammar used by common programming languages.

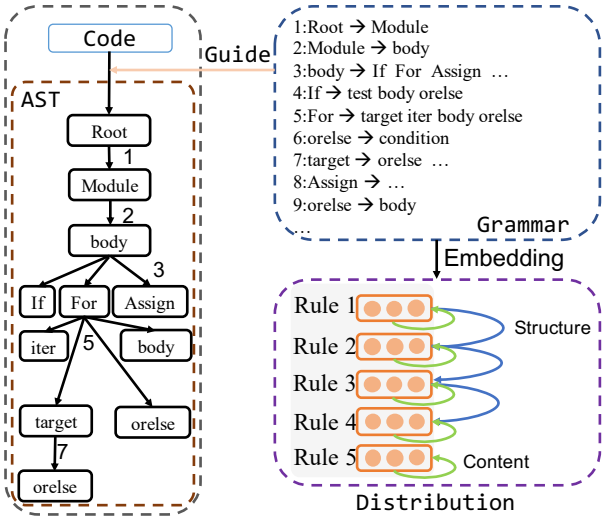

Figure 1: Context-free grammar and the corresponding AST.

sponds to a rule in the grammar, and each edge represents a parent-child relationship between two rules. Then, we use a graph neural network (GNN) to map each node into a vector. To preserve the content information, we further propose to use a gating layer to integrate the rule definition with the node embedding in the GNN. The above neural structure is then used as an embedding layer in existing neural network for embedding grammar rules.

Our experiment was conducted on six widely-used benchmarks containing four context-free languages. We used TreeGen [Sun *et al.*, 2020], one of the state-of-the-art models on these benchmarks, as our base model. We directly replace the embedding layer of TreeGen with Grape. The experimental results show that our approach improves the accuracy of TreeGen by 0.8-6.4 percentage points on six mainstream benchmark sets. Although our approach is designed for rule embedding, it also could be applied to AST token embedding. Thus, we combined Grape with Sandwich Transformer [Hellendoorn *et al.*, 2020] on method naming to demonstrate the generality of Grape. Grape also improves the base model by 1.6 F1 score. In summary, we make the following contributions:

- We make the first attempt to learn the embedding of grammar. We design a graph structure to represent context-free grammar and use a novel neural architecture to encode the graph.
- We evaluated our approach on six benchmarks containing four context-free languages. The experimental results show that our approach is effective in learning the syntax and semantic information of the grammar and achieved 0.8-6.4 percentage points improvement over TreeGen on these benchmarks. We also evaluated our approach for an additional code-related task, method naming, to show the generality of Grape. The experimental results show that Grape also improves the performance of base models by 1.6 F1 score.

2 Proposed Approach

Figure 2 shows the overview of Grape. As mentioned before, Grape takes the context-free grammar as input and outputs the embedding of each rule. To implement this component, we first map the grammar into a novel graph structure and then use a graph neural network to produce the embeddings.

2.1 Grammar Relation Graph

In this section, we introduce the detailed structure of the graph and the approach mapping a context-free grammar into the graph. We define a context-free grammar as $G^{(\text{gra})} = \langle N, T, R, \lambda \rangle$, where N denotes a set of non-terminals, T denotes a set of terminals, R denotes a set of production rules, and $\lambda \in N$ denotes a special start symbol. Especially, a production rule that can be applied to non-terminal A is represented as $A \to B_1\ B_2\ \cdots\ B_n$, and an application of the rule replaces A with the sequence $B_1\ B_2\ \cdots\ B_n$.

We propose *grammar relation graph* to represent context-free grammars. Formally, given a context-free grammar $G^{(\text{gra})}$, a grammar relation graph is a tuple $G = \langle V, E \rangle$, where $V = R$ denotes the vertexes in the graph, and $E \subseteq V \times V$ denotes the edges and is a minimal set satisfying the following condition: for two arbitrary rules $r_1 = A_1 \to B_{11}\ B_{12}\ \cdots\ B_{1m} \in V$ and $r_2 = A_2 \to B_{21}\ B_{22}\ \cdots\ B_{2n} \in V$, $(r_1, r_2) \in E$ if $A_2 = B_{1j}$ for some j such that $1 \le j \le m$.

Figure 2 shows an example of a grammar relation graph. On the left side, there is a set of production rules, where the leading numbers are the IDs of the production rules. On the right hand side there is the corresponding *grammar relation graph* of the given partial context-free grammar. Each node in the graph corresponds to a production rule in the grammar and is labeled with the ID of the rule. As shown, the rule 2 can be used to expand a non-terminal produced by rule 1, and thus node 1 has a directed edge to node 2 in the graph.

2.2 Neural Mechanism

The second part of Grape is a graph neural network (GNN). The model takes the grammar relation graph as input and outputs the vector of each rule. The overview of this part is shown in Figure 2. In this section, we will introduce the detailed structure of the GNN used in our approach.

Embedding Layer. The first layer of Grape is a one-hot encoding layer similar to the most base models. Grape transforms the unique ID of each production rule into a fixed-size vector for GNN to compute.

Static Encoding. We first encode each node in the grammar relation graph into a fixed-sized vector for GNN. Grape directly uses the ID of each production rule as the one-hot encoding label. Thus, we can encode each node in G as $\boldsymbol{n}_1, \boldsymbol{n}_2, \cdots, \boldsymbol{n}_P$ by table-lookup embeddings, where P denotes the total number of rules in R. We use the static encoding of r_i, as the initial embedding of the GNN.

Graph Neural Network. The computation of GNN iteratively updates the vector of nodes. In each iteration, the vector of each node is updated by combining those of its neighbors. The computation of each iteration is performed through two components. The first component, the definition encoding component, integrates the corresponding rule definition.

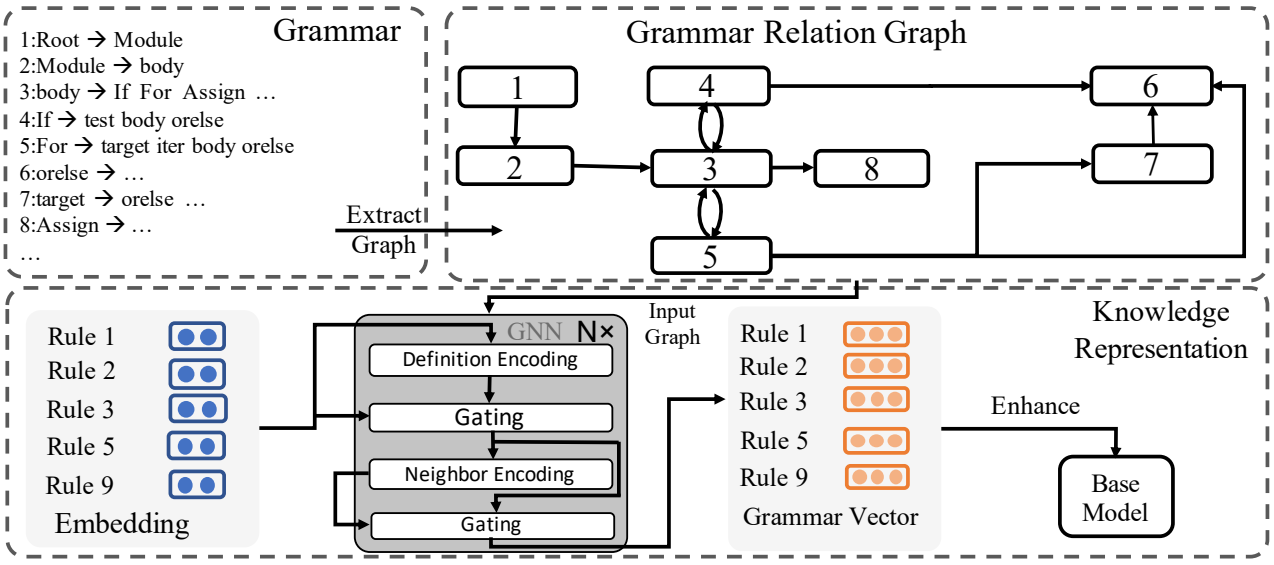

Figure 2: Context-free grammar and its grammar relation graph

The second component, the neighbor encoding component, combines the vector of each node with those of its neighbors.

Definition Encoding Component. As mentioned before, the static encoding of each node encodes only the ID of the rule. This component integrates the rule content into the vertex of each node. It first encodes the rule content and then uses a gating sub-layer to combine the encoded rule content with the vector of the node.

Definition Encoding Sub-Layer. To encode the content information of each production rule, we first represent the symbol in $T \bigcup N$ as a vector $\boldsymbol{s}$ through table-lookup embeddings. Then, for a production rule $r : \alpha \rightarrow \beta_1\ \beta_2 \cdots \beta_n$, we adopt a full-connected layer on the corresponding token embeddings. The rule encoding of r is computed as $\boldsymbol{s}_r = \boldsymbol{W}[\boldsymbol{s}_\alpha; \boldsymbol{s}_{\beta_1}; ...; \boldsymbol{s}_{\beta_n}]$, where $\boldsymbol{s}_\alpha$ and $\boldsymbol{s}_{\beta_j}$ are the table-lookup embeddings of the parent node α and the child node β_j, and $\boldsymbol{W}$ is a trainable parameter. The sequence of the child nodes is padded to a pre-defined maximum length. These vectors are then fed to the gating sub-layer, and are integrated with the input features for each node at each iteration.

Gating Sub-Layer. The input features for each node contain rich structural information of G after several iterations. To emphasize the importance of these features, we adopt a mechanism named Gating Mechanism [Sun *et al.*, 2020] to integrate the input features with the rule content encoding. This mechanism takes three vectors named $\boldsymbol{q}, \boldsymbol{c}_1, \boldsymbol{c}_2$ as input and incorporate $\boldsymbol{c}_1$ with $\boldsymbol{c}_2$ through the multi-head mechanism treating $\boldsymbol{q}$ as the control vector. The detailed computation can be found in Sun *et al.*[2020].

For the t-th iteration, we use the input feature $\boldsymbol{r}_t^i$ of the node r_i as the control vector to integrate itself with the corresponding definition encoding $\boldsymbol{s}_{r_i}$. Thus, the computation of node r_i at the t-th iteration can be represented as $\boldsymbol{m}_t^i = \mathrm{Gating}(\boldsymbol{r}_t^i, \boldsymbol{r}_t^i, \boldsymbol{s}_{r_i})$. The vector $\boldsymbol{m}_t^i$ is then fed to the second neural component to integrate the neighbors' information.

Neighbor Encoding Component. As the traditional GNN, the encoding of each node should be integrated with the encoding of its neighbors to extract the structural information of the graph. This component first computes the encoding of the neighbors through the adjacent matrix, and then uses a gating mechanism to combine the vector of the node with the encodings of its neighbors.

Neighbor Encoding Sub-Layer. As the traditional GNN, the encoding the neighbors of node r_i at the t-th iteration can be computed as $\boldsymbol{p}_t^i = \sum_{r^j \in G} A^n_{r^i r^j} \boldsymbol{m}_t^j$. Here, A^n is a normalized adjacency matrix of G. We use the normal operation proposed by Kipf and Welling [2017]: $A^n = S_1^{-1/2} A S_2^{-1/2}$. Here A is the adjacency matrix of G, S_1 and S_2 are the diagonal matrices with summation of A in columns and rows. To encode the direction of edges, we define a hyperparameter $\beta (\beta > 1)$ to represent this information. For example, if node r_i has a directed edge connected to r_j, cell $A_{i,j}$ is 1 and cell $A_{j,i}$ is β.

Gating Sub-Layer. To integrate the neighbor encoding with the input vectors for each node, we apply the same sub-layer to the input vector r_t^i and the neighbor encoding p_t^i. We use the input vector r_t^i as the control vector $\boldsymbol{q}$ to incorporate these two vectors. This computation can be represented as $\boldsymbol{r}_{t+1}^i = \mathrm{Gating}(\boldsymbol{m}_t^i, \boldsymbol{m}_t^i, \boldsymbol{p}_t^i)$.

We apply the GNN layer of N iterations on the initial encoding of each node and get vectors of each production rule. Then these vectors are fed to a base model.

3 Evaluation

3.1 Experiment I: Grammar Preserving Rule Embedding

We have implemented Grape for several different benchmarks in several context-free languages. In this section, we report our experiments and the performance of Grape.

	Code Generation			Semantic Parsing		Regex Synthesis
Statistics	**HS**	**DJANGO**	**CONCODE**	**ATIS**	**JOB**	**StrReg**
# Train	533	16,000	100,000	4,434	640	2173
# Dev	66	10,000	2,000	491	-	351
# Test	66	1,805	2,000	448	140	996
Avg. Token (NL)	35.0	10.4	71.9	10.6	8.7	33.5
Avg. Token (Code)	83.2	8.4	26.3	33.9	17.9	15.1
Node of Graph	772	668	477	180	50	193
Avg. Degree	3.81	3.55	4.20	2.80	2.28	5.52
Med. Degree	3	2	5	2	1	4

Table 1: Statistics of the datasets we used.

Dataset. We evaluated our approach on six benchmarks, including the HearthStone benchmark [Ling *et al.*, 2016], two semantic parsing benchmarks [Dong and Lapata, 2016], the Django benchmark [Yin and Neubig, 2017], the Concode benchmark [Iyer *et al.*, 2018] and the StrReg benchmark [Ye *et al.*, 2020]. The statistics of these datasets are shown in Table 1.[2]

The HearthStone benchmark contains 665 different cards of HearthStone. Each card is composed of a NL specification and a program written in Python. When processing the NL, we used the structural preprocessing as described in Sun *et al.* [2020]. The semantic parsing task contains two benchmarks. The input of this task is the NL specification, while the output is a short piece of lambda expressions in a specific DSL. The Django benchmark contains 18,805 lines of Python source code extracted from the Django web framework. Each line of code is annotated with a NL specification. The Concode benchmark contains 104,000 pairs of Java code and a NL specification with the programmatic context. The StrReg benchmark contains 3520 pairs of structurally complex and realistic regexes and a NL description.

The benchmarks involve four context-free languages: Java, Python, lambda expressions, and regular expressions. For Java and Python, we use the grammar extracted from their official parsers[3]. For lambda expressions, we used the grammar written by Kwiatkowski *et al.* [2013]. For regular expression, we use the grammar defined by Ye *et al.* [2020].

Metric and Hyperparameters. Existing studies use different metrics for different benchmarks. StrAcc, Acc+, ExeAcc, and DFAAcc all measure the percentage of correct programs, but use different definitions for correctness. StrAcc [Yin and Neubig, 2017] considers a program as correct when it has exactly the same token sequence as the ground truth. Acc+ [Sun *et al.*, 2019] further allows the renaming of variables. ExeAcc [Dong and Lapata, 2016] further considers the symmetry of operators. DFAAcc [Ye *et al.*, 2020] considers a regex is correct when it is DFA-equivalent compared with the groundtruth. To compare with existing results, we followed the metrics settings in existing studies.

For the hyperparameters of our model, we set the number of iterations $N = 9$. The hidden sizes were all set to 256. We applied dropout after each iteration of the GNN layer, where the drop rate is 0.15. The model was optimized by Adam with learning rate $lr = 0.0001$. We selected the hyperparameters based on the performance of validation set of Concode. The number of iterations has slight influence on the results. As we change the number from 6 to 11, the change of model performance on Concode is less than 0.4%. We run Grape five times with different random seeds on Atis, Job and HearthStone due to the small validation set of these benchmarks. At inference time, we used beam search with beam size $b = 5$ following Sun *et al.* [2019].

Base model and Inference. We use TreeGen [Sun *et al.*, 2020], which is one of the current state-of-art model on these benchmarks, as the base model. It uses a tree-based transformer to generate the rule sequence. Specially, Treegen chooses the next production rule among all possible candidates by softmax based on the outputs. In our approach, the embeddings of production rules contain rich structural and content information. Thus, we introduce the pointer network that directly selects a rule from the grammar relation graph. The pointer network is computed by

$$\theta = \boldsymbol{v}^T \mathbf{tanh}(W_1 \boldsymbol{h} + W_2 \boldsymbol{r}) \tag{1}$$

$$\mathrm{P}(\text{select rule s in step i}|\ \cdot\ ) = \frac{exp\{\theta_s\}}{\sum_{j=1}^{N_r} exp\{\theta_j\}} \tag{2}$$

where $\boldsymbol{h}$ denotes the outputs of decoder, $\boldsymbol{r}$ denotes the outputs of Grape fed into the base model.

For training, we first construct the grammar relation graph based on the grammar of the specific language. We then build the corresponding GNN layer based on the grammar graph and connect it with the base model. Thus, it is an end-to-end training where the output of GNN is directly fed into TreeGen and the gradients can be passed backwardly.

Overall Result. Table 2 shows the performance of our approach on several benchmarks, in comparison with previous state-of-the-art models. Each line in Table 2 corresponds to an existing approach and shows its performance. The first part denotes the traditional approaches using rule-based translation but not neural models. The third part denotes the neural models pre-trained with huge extra data. As shown, our approach boosts the performance of TreeGen on all benchmarks. In particular, the performance on Atis is even better than the traditional approaches. To the best of our knowledge, this is the first time that a neural approach has outperformed the traditional approaches on this benchmark. Furthermore, Grape achieves 6.4 points improvement over TreeGen on StrReg, which is the highest among these benchmarks. We conjecture the reason is that the training set of StrReg does not contain rich usage patterns of the grammar. When the training set contains rich usage patterns, a neural network may learn the structural and content information from it, but when the usage patterns are not rich, encoding grammar definitions become critical. These results suggest that learning the grammar embedding is effective in boosting the performance of TreeGen.

Error Rate of Predicted Rules. To understand whether Grape helps the base model to predict the production rule, we

[2]The code is available at https://github.com/pkuzqh/Grape

[3]The links of the parsers are https://docs.python.org/3/library/ast.html and https://github.com/c2nes/javalang for Python and Java.

		Code Generation					Semantic Parsing		Regex Synthesis
	Method	**HearthStone**			**Django**	**Concode**	**Atis**	**Job**	**StrReg**
	Metric	**StrAcc**	**BLEU**	**Acc+**	**StrAcc**	**StrAcc**	**ExeAcc**	**ExeAcc**	**DFAAcc**
	KCAZ13 [Kwiatkowski *et al.*, 2013]	-	-	-	-	-	89.0	-	-
	WKZ14 [Wang *et al.*, 2014]	-	-	-	-	-	91.3	90.7	-
Neural Networks	SEQ2TREE [Dong and Lapata, 2016]	-	-	-	-	-	84.6	90.0	-
	ASN+SUPATT [Rabinovich *et al.*, 2017]	22.7	79.2	-	-	-	85.9	**92.9**	-
	TRANX [Yin and Neubig, 2018]	-	-	-	73.7	-	86.3	90.0	-
	Iyer-Simp+200 idoms [Iyer *et al.*, 2018]	-	-	-	-	12.20	-	-	-
	GNN-Edge [Shaw *et al.*, 2019]	-	-	-	-	-	87.1	-	-
	SoftReGex [Park *et al.*, 2019]	-	-	-	-	-	-	-	28.2
	TreeGen [Sun *et al.*, 2020]	30.3±1.061	80.8	33.3	76.4	16.6	89.6±0.329	91.5±0.586	22.5
	GPT-2 [Radford *et al.*, 2019]	16.7	71	18.2	62.3	17.3	84.4	92.1	24.6
	CodeGPT [Lu *et al.*, 2021]	27.3	75.4	30.3	68.9	**18.3**	87.5	92.1	22.49
	TreeGen + Grape	**33.6±1.255**	**85.4**	**36.3**	**77.3**	17.6	**92.16±0.167**	**92.55±0.817**	**28.9**

Table 2: Performance of our model in comparison with previous state-of-the-art results on several benchmarks.

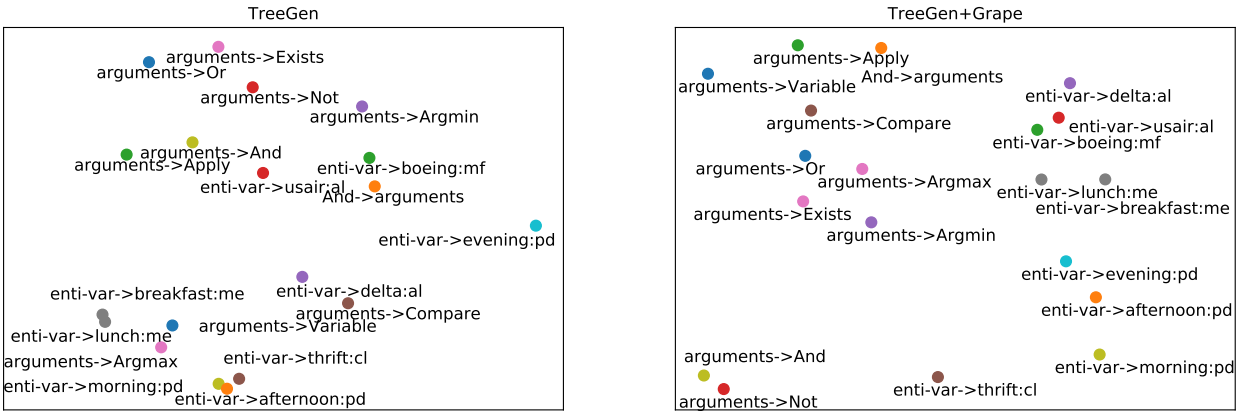

Figure 3: Visualizing some rule vectors in the DSL for semantic parsing with t-SNE

Model	**Top-1**	**Top-3**	**Top-5**
TreeGen	1.11%	31.61%	39.60%
TreeGen + Grape	**0.15%**	**27.09%**	**34.87%**

Table 3: Error rate of Models on Atis.

count the error rate[4] that the predicted production rule violates the grammar constraints on Atis benchmark as shown in Table 3. We count the error rate of the top-1, top-3, and top-5 predicted production rules by these models, which is defined by the number of grammatically incorrect predictions in top-k divided by the the total predictions. TreeGen with Grape all performs better compared with TreeGen. Especially, TreeGen with Grape achieves 0.15 percentage error rate of the top-1 prediction. These results show that Grape helps the base model to learn the constraint of the grammar.

Visualization of Rule Embeddings. To figure out whether Grape preserves the structural and content information of the grammar, we use the t-Distributed Stochastic Neighbor Embedding (t-SNE) model, a dimensionality reduction technique, to map the original vectors to 2-D space. Due to the space limit, we randomly choose several rules from the language for semantic parsing to show in the picture. As shown in Figure 3, rules with similar content tend to have similar representations. The rules which are relevant to node "arguments" mainly lie on the left side while the rules associated with "enti-var" lie on the right side while the rule encoding of TreeGen does not has this property. Also, rules with similar structure have similar representations. The rules that can have a common parent, such as $entivar \rightarrow evening : pd, entivar \rightarrow afternoon : pd$ and $entivar \rightarrow morning : pd$, have similar representations with Grape in the right sub-figure. However, $entivar \rightarrow evening : pd$ lies far away from $entivar \rightarrow morning : pd$ without Grape in the left sub-figure. These observations suggest that Grape leads the model to learn a better embedding of production rules.

Time Efficiency and Complexity. We further evaluated the complexity of our model. It takes 34.78s for an epoch on a single Nvidia Titan RTX with Grape on average, whereas 30.74s without Grape. And Grape has 6.5×10^5 parameters on average.

[4]A prediction is incorrect if there are grammatically incorrect rules in the top-k predicted rules. The error rate of the top-k denotes the rate of incorrect prediction during generating.

Metric	Precision	Recall	F1
Code2Seq [Alon *et al.*, 2018]	50.64	37.40	43.02
Code2Vec [Alon *et al.*, 2019]	18.51	18.74	18.62
Transformer [Vaswani *et al.*, 2017]	38.13	26.70	31.41
S-Transformer [Hellendoorn *et al.*, 2020]	52.64	48.08	50.25
S-Transformer+Grape	**53.22**	**50.55**	**51.85**

Table 4: Performance of our approach on Method Naming.

3.2 Experiment II: Grammar Preserving AST Token Embedding

Apart from the production rule sequence, we observe that several approaches [Alon *et al.*, 2019; Sun *et al.*, 2019], use the AST traverse sequence to represent the CFS. These approaches usually treat the AST tokens as words and adopt word embedding to represent the AST tokens. Although Grape is designed for rule embedding, Grape also can be applied to represent the AST tokens. To adapt Grape for the AST token embedding, we replace the corresponding rule $r_i = \alpha \rightarrow \beta_1\ \beta_2 \cdots \beta_n$ of i-th node with α in the grammar relation graph and merge the nodes which have the same label in the graph.

To evaluate the effectiveness, we conducted an additional experiment on method naming task. In this task, the model needs to predict the name of a method given its corresponding body. Following Alon *et al.* [2018], the model predicts the target method name as a sequence of sub-tokens, e.g., getIndexOf is predicted as the sequence "get index of". As described by Hellendoorn *et al.* [2020], Sandwich Transformer (S-Transformer) adopts Transformer to extract the long dependency of the code and adopts GNN layers to learn the structural information of the code. Thus, we used S-Transformer, which is capable of handling long dependencies and rich structural information, as our base model.

For this task, we adopted the widely used Java benchmark [Alon *et al.*, 2019; Alon *et al.*, 2018], *Java-small*, which contains 11 relatively large Java projects. We took 9 projects for training, 1 project for validation and 1 project for test following Alon *et al.* [2018]. This dataset contains 691,607 examples in the training set, 23,844 examples in the validation set and 57,088 examples in the test set. We used three metrics, precision, recall, and F1 score over the target sequence as Alon *et al.* [2019]. We used F1 score on the development set to select the best model.

The performance of Grape on method naming task is shown in Table 4. Each line in Table 4 shows the performance measured by the three metrics on the benchmark of the corresponding approach. Compared with S-Transformer, Grape improves the performance on three metrics by 0.58, 2.47, 1.60, respectively. These results exhibit the generality of Grape.

4 Related Work

Code Generation. Code generation aims to generate code from a NL specification [Ling *et al.*, 2016; Dong and Lapata, 2016; Liang *et al.*, 2022] and has been intensively studied during recent years. The early work generates code based on templates and searching [Kwiatkowski *et al.*, 2013; Wang *et al.*, 2014]. Ling *et al.* [2016] proposed a sequence-to-sequence framework to generate code as sequences. Note that code has the constraints of grammar, which is different from NL. Thus, the abstract syntax tree (AST) was used to represent the generated code [Dong and Lapata, 2016; Yin and Neubig, 2017; Rabinovich *et al.*, 2017; Xiong *et al.*, 2018]. To alleviate the long dependency problem, Sun et al. applied the convolutional neural network [Sun *et al.*, 2019] and transformer [Sun *et al.*, 2020] for code generation. However, these existing approaches just embed the partial rule list through static labeling directly. Our approach further integrate the structural and content information of the grammar into the embeddings.

Distributed representations. Learning distributed representation for elements (e.g. words, code tokens, AST tokens) have been used to boost the performance of the neural models in several areas of natural language processing [Pennington *et al.*, 2014; **?**; Alon *et al.*, 2019]. **?** [**?**] introduced, word2vec, an approach to learning a good representation of words based on self-supervised tasks. Devlin *et al.* [2018] proposed, BERT, to compute the word embedding with the sentence context via Transformer [Vaswani *et al.*, 2017]. Inspired by this work, [Alon *et al.*, 2019] used a path-attention based model to learn the representation of code. In this paper, we propose Grape to learn a grammar-preserving rule embedding via a well-designed GNN.

Graph Neural Network. Graph Neural Network has been widely investigated in recent years. The concept of GNN was proposed byC Scarselli *et al.*. Based on this framework, various GNNs have been proposed [Kipf and Welling, 2017; Li *et al.*, 2016; Zhong and Mei, 2020]. In particular, Li *et al.* [2016] proposed a gated graph neural network(GGNN), which adapts the LSTM layer to GNN for better encoding the node.

5 Conclusion

In this paper, we propose an approach to learn the distributed representation of context-free grammar. We first introduce a novel graph to represent the grammar. Then we adopt a gated graph neural network to extract the structural information of production rules and integrate the content information with the node embedding via a gating layer. To confirm the effectiveness of our approach, we conducted experiments on several widely used benchmarks on code generation and semantic parsing. The results show that Grape learns a good embedding of the production rules and the combined model surpasses the existing state-of-the-art methods on these benchmarks.

Acknowledgements

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2019YFE0198100, the Innovation and Technology Commission of HKSAR under Grant No. MHP/055/19, National Natural Science Foundation of China under Grant No. 61922003 and a grant from ZTE-PKU Joint Laboratory for Foundation Software.

References

[Alon *et al.*, 2018] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *ICLR*, 2018.

[Alon *et al.*, 2019] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *POPL*, January 2019.

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

[Dong and Lapata, 2016] Li Dong and Mirella Lapata. Language to Logical Form with Neural Attention. In *ACL*, pages 33–43, 2016.

[Hellendoorn *et al.*, 2020] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *ICLR*, 2020.

[Iyer *et al.*, 2018] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *EMNLP*, pages 1643–1652, Brussels, Belgium, October-November 2018.

[Kipf and Welling, 2017] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[Kwiatkowski *et al.*, 2013] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. In *EMNLP*, pages 1545–1556, 2013.

[Li *et al.*, 2016] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *ICLR*, 2016.

[Liang *et al.*, 2022] Qingyuan Liang, Qihao Zhu, Zeyu Sun, Lu Zhang, Wenjie Zhang, Yingfei Xiong, Guangtai Liang, and Lian Yu. A survey of deep learning based text-to-sql generation (in chinese). *SCIS*, page 1–30, 2022.

[Ling *et al.*, 2016] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. Latent Predictor Networks for Code Generation. In *ACL*, pages 599–609, 2016.

[Lu *et al.*, 2021] Shuai Lu, Daya Guo, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS*, 2021.

[Mikolov *et al.*, 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[Mou *et al.*, 2016] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, pages 1287–1293, 2016.

[Park *et al.*, 2019] Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. Softregex: Generating regex from natural language descriptions using softened regex equivalence. In *EMNLP-IJCNLP*, pages 6425–6431, 2019.

[Pennington *et al.*, 2014] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[Rabinovich *et al.*, 2017] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*, pages 1139–1149, 2017.

[Radford *et al.*, 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[Scarselli *et al.*, 2009] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *TNN*, 20(1):61–80, 2009.

[Shaw *et al.*, 2019] Peter Shaw, Philip Massey, Angelica Chen, Francesco Piccinno, and Yasemin Altun. Generating logical forms from graph representations of text and entities. In *ACL*, pages 95–106, July 2019.

[Sienčnik, 2015] Scharolta Katharina Sienčnik. Adapting word2vec to named entity recognition. In *NODALIDA*, pages 239–243, 2015.

[Sun *et al.*, 2019] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural cnn decoder for code generation. In *AAAI*, volume 33, pages 7055–7062, 2019.

[Sun *et al.*, 2020] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. In *AAAI*, volume 34, pages 8984–8991, 2020.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 6000–6010, 2017.

[Wang *et al.*, 2014] Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. Morpho-syntactic lexical generalization for ccg semantic parsing. In *EMNLP*, pages 1284–1295, 2014.

[Wolf *et al.*, 2014] Lior Wolf, Yair Hanani, Kfir Bar, and Nachum Dershowitz. Joint word2vec networks for bilingual semantic representations. *IJCLA*, 5(1):27–42, 2014.

[Xiong *et al.*, 2018] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. Learning to Synthesize. In *GIW*, 2018.

[Ye *et al.*, 2020] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Benchmarking multimodal regex synthesis with complex structures. In *ACL*, pages 6081–6094, 2020.

[Yin and Neubig, 2017] Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*, pages 440–450, 2017.

[Yin and Neubig, 2018] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. 10 2018.

[Zhong and Mei, 2020] Hao Zhong and Hong Mei. Learning a graph-based classifier for fault localization. *SCIS*, 63, 06 2020.