The impact of lexical and grammatical processing on generating code from natural language

Anonymous ACL submission

Abstract

Considering the seq2seq architecture of Yin and Neubig (2018) for natural language to code translation, we identify four key components of importance: grammatical constraints, lexical preprocessing, input representations, and copy mechanisms. To study the impact of these components, we use a state-of-the-art architecture that relies on BERT encoder and a grammar-based decoder for which a formalization is provided. The paper highlights the importance of the lexical substitution component in the current natural language to code systems.

1 Introduction

Translating natural language program descriptions to actual code is meant to help programmers to ease writing reliable code efficiently by means of a set of advanced code completion mechanisms.

There are mainly two classes of methods for obtaining code corresponding to a query expressed in natural language. The first one is code retrieval, which consists of searching and retrieving an appropriate code snippet from a code database. The second one is code generation, where the goal is to generate code fragments from a natural language description, generating potentially previously unseen code. In this work, we are interested in Python code generation. Code generation features a mismatch between an ambiguous and noisy natural language input and the structured nature of the generated code. Although Python’s vocabulary has a finite number of keywords, the set of value that can be assigned to a variable is infinite and constitutes one of the issues in predicting code corresponding to natural language.

Like many other NLP tasks, current architectures for natural language to code generally take advantage of pre-trained language models such as BERT (Devlin et al., 2019) or GPT (Brown et al., 2020) based on the transformer architecture (Vaswani et al., 2017). In particular, these architectures are used for code generation where parallel data is limited due to the human expertise required for alignment. The best results on code generation are reached by pretraining seq2seq models on external sources, then by fine-tuning those models on smaller data sets. For instance, Orlanski and Gittens (2021) fine-tunes BART (Lewis et al., 2020) on data pairs of natural language and code and by taking advantage of external informations. Similarly, Norouzi et al. (2021) used BERT and a transformer decoder in a semi-supervised way by taking advantage of a large amount of additional monolingual data. Another popular method is to train large language models on code (Austin et al., 2021; Hendrycks et al., 2021). Notably, GPT-3 has been finetuned on a large quantity of data from Github to obtain a powerful language model named Codex (Chen et al., 2021) that powers Github Copilot, a tool to help developers.

Overall the above mentioned solutions aim to take advantage of large amounts of training data available nowadays, but few of them care about generating code that is guaranteed to be syntactically correct nor well typed. Let us mention some exceptions from semantic parsing like Dong and Lapata (2016); Rabinovich et al. (2017); Yin and Neubig (2017) that rely on grammatical constraints to ensure that the generated code can be executable.

In this work, we study variations around the TranX seq2seq architecture (Yin and Neubig, 2018) for translating natural language to code. Rather than generating directly code tokens from natural language, the architecture generates an Abstract Syntax Tree (AST) constrained by the programming language grammar.

The paper reports state of the art results on the task and specifically introduces:

• A formalization of the grammar constrained code generator relying on the Earley (1970) parser transition system.
• A study of the impact of key components of the architecture on the performance of the system: we study the impact of the grammatical component itself, the impact of the language model chosen, the impact of variable naming and typing and the impact of the input/output copy mechanisms.

It is structured as follows. Section 2 formalizes the symbolic transition system used for generating the grammatically correct code, Section 3 describes a family of variants around the TranX architecture that will be used to study the impact of these variations in the experimental part of the paper (Section 4).

2 A transition system for code generation

Among the models tested in the paper, some are generating syntactically constrained code. In the context of our study, we propose a transition model that meets two objectives: the code generated is grammatically valid in terms of syntax and the whole translation process still reduces to a seq2seq transduction mechanism that allows us to leverage standard machine learning methods.

To this end we introduce a transition system for code generation that generates an AST as a sequence of actions. The derivations can then be translated into ASTs and in actual Python code by means of deterministic functions. The set of valid ASTs is a set of trees that are generated by an ASDL grammar (Wang et al., 1997). An ASDL grammar is essentially a context free grammar abstracting away from low level syntactic details of the programming language and aims to ease the semantic interpretation of the parse trees. To this end ASDL grammar rules come with additional decorators called constructors and field names (Figure 1).

Our transition system generates derivations, or sequences of actions, that can be translated to a syntactically correct Python code. We adapt to code generation the transition system of the Earley parser (Earley, 1970) as formalized in Figure 2. The generator state is a stack of dotted rules. A dotted rule is a rule of the form $A \rightarrow \alpha \cdot X \beta$ where $\alpha$ is a sequence of grammar symbols whose subtrees are already generated and $X \beta$ is a sequence of grammar symbols for which the subtrees are yet to be generated. The $\cdot X$ symbol is the dotted symbol or the next symbol for which the system has to generate the subtree. The Python ASDL grammar includes rules with star ($\ast$) qualifiers allowing zero or more occurrences of the starred symbol. The transition system uses an additional set of starred actions and a $\text{CLOSE}$ action to stop these iterations (Figure 2).

Each $\text{PREDICT}(C)$ action starts the generation of a new subtree from its parent. The $\text{GENERATE}$ action adds a new leaf to a tree. The $\text{COMPLETE}$ action finishes the generation of a subtree and continues the generation process with its parent. The set of $\text{PREDICT}$ actions is parametrized by the ASDL rule constructor ($C$), thus there are as many predict actions as there are constructors in the ASDL grammar. Constructors are required in order to generate the actual ASTs from the derivations.

$\text{GENERATE}(V)$ actions are actions responsible for generating the terminal or primitive symbols. The Python ASDL grammar generates ASTs with primitive leaf types (identifier, int, string, constant) that have to be filled with actual values for the AST to be useful. To generate actual primitive values the set of generate actions is also parametrized by the actual values $V$ for the primitive types. The set of such values is infinite and consequently the set of generate actions is also infinite.

Non determinism comes from the use of $\text{PREDICT}(C)$, $\text{GENERATE}(V)$ and $\text{CLOSE}$ rules. By contrast the application of the $\text{COMPLETE}$ action is entirely deterministic: once the generator has a completed dotted rule on the top of its stack, it has no other choice than applying the complete rule.

The sequential generation process is illustrated in Figure 3. Given a start state, at each time step, the generator has to decide which action to perform according to the current state of the stack and updates the stack accordingly. Once the generator reaches the goal state, we collect the list of actions performed (the derivation) in order to build the AST that we finally translate into actual Python code.\(^1\)

3 Factors influencing code prediction

All architectures analyzed in this study are variations around a seq2seq architecture. We describe the several variants of this architecture used in this paper both on the encoder and decoder side. We identify key factors that have an impact on the natural-language-to-code translation architecture.

\(^1\)We use the astor\(^2\) library to this end.
Figure 1: Example of ASDL rules for the Python language. Each rule is built from a set of grammatical symbols (in blue), is uniquely identified by a constructor name (in red) and provides names to its right hand side symbols, its fields (in green). Grammatical symbols are split in nonterminals (like `expr`) and terminals or primitives (like `constant`). Grammatical symbols can also be annotated with qualifiers (`*`) that allow for zero or more iterations of the symbol.

```
expr  =  BinOp expr left, operator op, expr right
operator = Add
expr = Constant constant value
expr = List expr+ elt
```

Figure 2: An Earley inspired transition system for generating Abstract Syntactic Trees. The state of the generator is a stack of dotted rules whose bottom is $S$. As in the the Earley parser, the `PREDICT` rule starts the generation of a new subtree by pushing a new dotted rule on the stack, the `GENERATE` rule adds a leaf to the tree by swapping the top of the stack and the `COMPLETE` rule attaches a generated subtree into its parent by popping the top two elements of the stack and pushing an updated dotted rule. To handle `*` qualifiers we add the starred inference rules where `COMPLETE*` and `GENERATE*` implement an iteration that stops with the `CLOSE*` rule.

<table>
<thead>
<tr>
<th>Action</th>
<th>Transition</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>START(C)</td>
<td>$(A \rightarrow \bullet \alpha)$</td>
<td></td>
</tr>
<tr>
<td>GOAL</td>
<td>$(A \rightarrow \alpha \bullet)$</td>
<td></td>
</tr>
<tr>
<td>PREDICT(C)</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet B\beta) \Rightarrow (S</td>
</tr>
<tr>
<td>GENERATE(V)</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet \beta) \Rightarrow (S</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet B\beta</td>
</tr>
<tr>
<td>PREDICT* (C)</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet B^*\beta) \Rightarrow (S</td>
</tr>
<tr>
<td>GENERATE* (V)</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet t^*\beta) \Rightarrow (S</td>
</tr>
<tr>
<td>COMPLETE*</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet B^*\beta</td>
</tr>
<tr>
<td>CLOSE*</td>
<td>$(S</td>
<td>A \rightarrow \alpha \bullet X^*\beta) \Rightarrow (S</td>
</tr>
</tbody>
</table>

Figure 3: Example derivation for the generation of the Python list expression `[7+5, 4]`. The derivation starts with `expr` as axiom symbol and applies transitions until the goal is reached. The list of actions performed is called the generator `derivation`. Given a generated derivation we can design a straightforward deterministic procedure to translate it into an `AST`. The actual Python code is generated from the `AST` by the `astor` library.
and we formalize a family of models that allow to
test variations of these factors.

We consider a family of models generating
Python code $y$ from a natural language description
$x$, that have the generic form:

$$p(y|x) = \prod_{t} p(y_t|y_{<t}, x) \quad (1)$$

$y$ is either a sequence of code tokens in case we do
not use a grammar, or a sequence of actions from a
derivation in case we use a grammar. The decoding
objective aims to find the most-probable hypothe-
sis among all candidate hypotheses by solving the
following optimization problem:

$$\hat{y} = \arg\max_{y} p(y|x) \quad (2)$$

The family of models varies according to four
key qualitative factors that we identify in the TranX
architecture. First we describe a substitution proce-
dure managing variables and lists names in section
3.1). Second, in section 3.2, we test the architec-
tural variations for encoding the natural language
sequence. Third, in section 3.3, we describe vari-
ations related to constraining the generated code
with grammatical constraints and architectural vari-
ations that allow to copy symbols from the natural
language input to the generated code.

3.1 Substitution

Programming languages come with a wide range of
variable names and constant identifiers that make
the set of lexical symbols infinite. Rather than
learning statistics on a set of ad-hoc symbols, we
rather normalize variable and constant names with
a pre-processing method, reusing the method of
Yin and Neubig (2018).

Preprocessing amounts to substitute the actual
names of the variables with a normalized set of pre-
defined names known to the statistical model. The
substitution step renames all variables both in the
natural language and in the code with conventional
names such as var_0, var_1, etc. for variables
and lst_0, lst_1, etc. for lists. A post processing
step substitutes back the predicted names with the
original variable names in the system output.

For example, given the natural language intent:

\[
\text{create list `done` containing permutations of each element in list `\{a, b, c, d\}` with variable `x` as tuples}
\]

is transformed into:

\[
\text{create list var_0 containing permutations of each element in list lst_0 with variable var_1 as tuples}
\]

The predicted code such as $\text{var_0} = \{(\text{el}, \text{var_1}) \text{ for el in [lst_0]}\}$ is transformed back into $\text{done} = \{(\text{el}, \text{x}) \text{ for el in [a, b, c, d]}\}$.

Models using variable replacement as illustrated
above, are identified with the notation SUBSTITU-
TION = TRUE in section 4. Implementing this
heuristic is made easy by the design of the CoNaLa
data set where all such names are explicitly quoted
in the data while for Django we had to define our
own heuristic.

3.2 Encoder

We switched between a classic bi-LSTM and a
pretrained BERT\textsubscript{BASE} to encode the input natural
language $\{x_i, i \in [1, n]\}$ of $n$ words into a vectorial representations $\{h_{i}^{(enc)}, i \in [1, n]\}$ which are
later used to compute the attention mechanism.
We set the BERT factor to TRUE when using it and
FALSE when using the bi-LSTM.

3.3 Decoder

At each time step $t$, the LSTM decoder computes
its internal hidden state $h_{t}^{(dec)}$:

$$h_{t}^{(dec)} = \text{LSTM}(e_{t-1} : \tilde{a}_{t-1}, h_{t-1}^{(dec)}) \quad (3)$$

where $e_{t-1}$ is the embedding from the previous
prediction, $\tilde{a}_{t-1}$ is the attentional vector.

We compute the attentional vector $\tilde{a}_t$ as in Lu-
ong et al. (2015) combining the weighted average
over all the source hidden state $c_t$ and the decoder
hidden state $h_{t}^{(dec)}$:

$$\tilde{a}_t = W_a[c_t : h_{t}^{(dec)}] \quad (4)$$

It is the attention vector $\tilde{a}_t$ which is the key to
determine the next prediction $y_t$.

We use several variants of the code generator,
that we describe by order of increasing complexity.
The basic generator is a feed forward that uses the
attention vector to generate a code token $v$ from a
vocabulary $V$:

$$p(y_t = \text{GENERATE}[v]|x, e_{<t}) = \text{softmax}(e_v^T \cdot W_g \cdot \tilde{a}_t) \quad (5)$$

These models are not constrained by the Python
grammar and we identify these models with GRAM-
MAR = FALSE.
We also use a pointer network that may either copy symbols from input to output or generate symbols from V. Then the probability of generating the symbol v is given by the marginal probability:

\[
p(y_t = \text{GENERATE}[v]|x, e_{<t}) = \\
p(\text{gen}|x, e_{<t})p(v|\text{gen}, x, e_{<t}) + p(\text{copy}|x, e_{<t})p(v|\text{copy}, x, e_{<t})
\]

(6)

The probabilities \(p(\text{gen}|.)\) and \(p(\text{copy}|.)\) sum to 1 and are computed with \(\text{softmax}(W \cdot \tilde{a}_t)\). The probability of generating \(v\) from the vocabulary \(V\) \(p(v|\text{gen},.)\) is defined in the same way as (5).

We use the pointer network architecture (Vinyals et al., 2015) to compute the probability \(p(v|\text{copy},.)\) of copying an element from the natural language \(x\). Models that use a pointer network are identified with \(\text{PN} = \text{TRUE}\), otherwise with \(\text{PN} = \text{FALSE}\).

Finally we use a set of models that are constrained by the Python grammar and that rely on the transition system from section 2. Rather than directly generating Python code, these models generate a derivation whose actions are predicted using two prediction tasks.

When the generator is in a state where the dot of the item on the top of the stack points on a terminal symbol, the \textit{generate} task is used. This amounts to reuse either equation (5) or equation (6) according to the model at hand. Models constrained by the grammar are labelled with \(\text{GRAMMAR} = \text{TRUE}\). Recall that the \textit{COMPLETE} action of the transition system is called deterministically (Section 2).

4 Experiments

In this section we describe the characteristics of the data sets on which we have tested our different setups and the underlying experimental parameters\(^3\).

4.1 Data sets

In this study we use two available data sets, Django and CoNaLa, to perform our code generation task.

The Django data set provides line-by-line comments with code from the Django web framework. About 70% of the 18805 examples are simple

---

\(^3\)The code of our experiments is public and available at anonymized adress
Python operation ranging from function declarations to package imports, and including exception handling. Those examples strongly share the natural language structure (e.g. `call the function cache.close → cache.close()`). More than 26% of the words in the natural language are also present in the code, BLEU score between the natural language and code is equal to 19.4.

CoNaLa is made up of 600k NL-code pairs from StackOverflow, among which 2879 examples have been manually cleaned up by developers. All results are reported on the manually curated examples, unless stated otherwise. The natural language descriptions are actual developer queries (e.g. `Delete an element 0 from a dictionary 'a'`) and the associated code is diverse and idiomatic (e.g. `{i: a[i] for i in a if (i != 0)`). Compared to Django, the code is much more challenging to generate. Especially because the number of words shared between the NL and the code is much lower (BLEU = 0.32). Also, the code is longer and more complex with an AST depth of 7.1 on average against 5.1 for Django.

4.2 Vocabulary generation

The vocabulary of natural language and code is essential. Usually, this vocabulary is created by adding all the words present in the training data set. There are however exceptions that are detailed in this section.

The natural language vocabulary relies on a byte pair encoding tokenizer when BERT = TRUE. As explained in section 3.1, the variable names are replaced with special tokens var_i and lst_i. These new tokens are crucial to our problem, and added to the BERT vocabulary. We can then fine-tune BERT with this augmented vocabulary on our data sets.

For the decoder part, when GRAMMAR = TRUE, the vocabulary of grammatical actions is fixed, while the vocabulary of AST leaves has to be built. This associated vocabulary can be composed of built-in Python functions, libraries with their associated functions or variable names. Its creation is consequently a major milestone in the generation process.

To create this external vocabulary, we proceed as in TranX. From the code, we create the derivation sequence composed of the action of the grammar as well as the primitives. All primitives of the action sequences are incorporated into our external vocabulary.

4.3 Setup

When BERT = FALSE, the size of the representations is kept small to prevent overfitting. Encoder and decoder embedding size is set to 128. The hidden layer size of the encoder and decoder bi-LSTM is set to 256 and the resulting attention vector size is 300. We have two dropout layers: for embeddings and at the output of the attention. We use Adam optimizer with learning rate $\alpha = 5 \times 10^{-3}$.

When BERT = TRUE, encoder embeddings have a natural size of 756 with BERT. We therefore apply a linear transformation to its output to get an embedding size equal to 512. The size of LSTM decoder hidden state and attention vector are set to 512. We regularize only the attentional vector in that case. We use Adam optimizer with learning rate $\alpha = 5 \times 10^{-5}$. In both cases, we use a beam search size of 15 for decoding.

Evaluation

We report the standard evaluation metric for each data set: exact match accuracy and corpus-level BLEU.

Python version

As the grammar slightly changes between Python versions, let us mention that all our experiments have been carried out with Python 3.7.

4.4 Ablation study

![Figure 5: Difference between the marginal mean of each variable for the TRUE and FALSE conditions.](image)

To highlight the contribution of the different factors, SUBSTITUTION, BERT, GRAMMAR, PN on the Django and CoNaLa data sets we report a detailed study of their impact in Table 1.
The results are analyzed by distinguishing lexical and grammatical aspects and by identifying relations between the different factors. We start by a comparison of the marginal mean of the BLEU score for each of our variables in both conditions. Figure 5 highlights the mean difference between the conditions by contrasting the case where the value is True with the case where the value is False.

**Pointer network** The pointer network can improve the results, especially when SUBSTITUTION = FALSE. This is because the only way to obtain the name of the variables is to copy them. Combined with substitution, the pointer network offers an additional possibility to predict the var_i, lst_i which allows to achieve the best results with a BLEU score of 39.01 on CoNaLa and an exact match accuracy of 76 on Django.

**Substitution and Typing** The scores are stabilised and much higher with substitution. We gain more than 9 points of BLEU on CoNaLa (respectively 20 points on Django) thanks to substitution. The "weakest" configuration where all variables are FALSE except the substitution gives better results than all configurations where SUBSTITUTION = FALSE.

The increase in BLEU with substitution can be explained in two ways. On the one hand, we remark that the model has difficulties to memorize the values to fill the lists with GENERATE. For example, four tokens of code must be generated to predict the list [a, b, c, d]. Using substitution, the model can just predict lst_0 which will be replaced by [a, b, c, d] during postprocessing. This avoids a potential error in the creation of the list and directly gives a valid 4-gram. This contributes to greatly increase the BLEU, which shows the importance of replacing listf. On CoNaLa, BLEU score on the development set drops from an average of 37.99 to an average of 30.66 without list replacement. Besides list replacement, the architecture has also a weakness with respect to variable typing. When using the grammar without substitution, the results are lower than without grammar. This effect is the result of a type checking failure. The model predicts ill-typed AST structures. For instance it predicts an AST whose corresponding variables have the type int during postprocessing.

**Grammatical aspect** The transition system doesn’t improve the results on average because
We formalized a transition system that allows us to guarantee the generation of syntactically correct code. A detailed study of the components of the seq2seq architecture reveals that the models have difficulties at managing accurately variable names and list encodings. The comparison with models trained on larger noisy data sets reveals that our grammatically constrained architecture without explicit denoising remains competitive. This further highlights the importance of grammatical constraints and of specific processes dedicated to manage variables, list naming and typing.
References


### A Additional Qualitative Examples

We present examples of code generated by our best models with and without grammar.

<table>
<thead>
<tr>
<th>Source</th>
<th>convert tuple ‘t’ to list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>list(t)</td>
</tr>
<tr>
<td>Grammar</td>
<td>[x for x in t for x in t]</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>[i for i in t]</td>
</tr>
<tr>
<td>Remark</td>
<td>Problem of CLOSE for the Grammar output. Without grammar the code is correct but with a low BLEU.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>get the position of item 1 in ‘testlist’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>[i for i, x in enumerate(testlist) if x == 1]</td>
</tr>
<tr>
<td>Grammar</td>
<td>[i for i, v in enumerate(testlist) if v == 1]</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>testlist = [i for i in testlist if i != 1]</td>
</tr>
<tr>
<td>Remark</td>
<td>Grammar output is not equal to Gold due to dummy variable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>declare an array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>my_list = []</td>
</tr>
<tr>
<td>Grammar</td>
<td>x = [0] * 2</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>[(0) for _ in range (10000)]</td>
</tr>
<tr>
<td>Remark</td>
<td>Source is not precise enough. Models’ outputs are valid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>append a numpy array ‘b’ to a numpy array ‘a’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>np.vstack((a, b))</td>
</tr>
<tr>
<td>Grammar</td>
<td>a = numpy.array([b, a])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>activate is a lambda function which returns None for any argument x.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>activate = lambda x : None</td>
</tr>
<tr>
<td>Grammar</td>
<td>activate = lambda x = None : x</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>activate = lambda x : None</td>
</tr>
<tr>
<td>Remark</td>
<td>Good BLEU for grammar output while the result is not adequate.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>remove first element of text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>text = text[1:]</td>
</tr>
<tr>
<td>Grammar</td>
<td>text = text[1:]</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>text[1:]</td>
</tr>
<tr>
<td>Remark</td>
<td>Syntax mistake for the code without grammar.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>increment piece by first element of elt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>piece += elt[0]</td>
</tr>
<tr>
<td>Grammar</td>
<td>piece += elt[1]</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>piece += elt[1]</td>
</tr>
<tr>
<td>Remark</td>
<td>First element of a list is zero, not one.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>activate is a lambda function which returns None for any argument x.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>activate = lambda x : None</td>
</tr>
<tr>
<td>Grammar</td>
<td>activate = lambda x = None : x</td>
</tr>
<tr>
<td>Without Grammar</td>
<td>activate = lambda x : None</td>
</tr>
<tr>
<td>Remark</td>
<td>Good BLEU for grammar output while the result is not adequate.</td>
</tr>
</tbody>
</table>