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 $A \rightarrow \alpha \bullet X \beta$ where α 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 $\bullet X$ 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 ($*$) qualifiers allowing zero or more occurrences of the starred symbol. The transition system uses an additional set of starred actions and a CLOSE action to stop these iterations (Figure 2).

Each PREDICT(C) action starts the generation of a new subtree from its parent. The GENERATE action adds a new leaf to a tree. The COMPLETE action finishes the generation of a subtree and continues the generation process with its parent. The set of 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.

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 PREDICT(C), GENERATE(V) and CLOSE rules. By contrast the application of the 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.

```
expr      =  BinOp expr left, operator op, expr right
operator  =  Add
expr      =  Constant constant value
expr      =  List expr* elts
```

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 ($\star$) that allow for zero or more iterations of the symbol.

Action	**Transition**			**Condition**
START(C)	$\langle A \to \bullet\alpha\rangle$			
GOAL	$\langle A \to \alpha\bullet\rangle$			
PREDICT(C)	$\langle \mathbf{S} \mid A \to \alpha \bullet B\beta\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha \bullet B\beta \mid B \to \bullet\gamma\rangle$	$(B \to \gamma \in \text{rules})$
GENERATE(V)	$\langle \mathbf{S} \mid A \to \alpha \bullet t\beta\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha t \bullet \beta\rangle$	$(t \in \text{primitives})$
COMPLETE	$\langle \mathbf{S} \mid A \to \alpha \bullet B\beta \mid B \to \gamma\bullet\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha B \bullet \beta\rangle$	
PREDICT*(C)	$\langle \mathbf{S} \mid A \to \alpha \bullet B^*\beta\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha \bullet B^*\beta \mid B \to \bullet\gamma\rangle$	$(B \to \gamma \in \text{rules})$
GENERATE*(V)	$\langle \mathbf{S} \mid A \to \alpha \bullet t^*\beta\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha t^\bullet t^*\beta\rangle$	$(t \in \text{primitives})$
COMPLETE*	$\langle \mathbf{S} \mid A \to \alpha \bullet B^*\beta \mid B \to \gamma\bullet\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha B \bullet B^*\beta\rangle$	
CLOSE*	$\langle \mathbf{S} \mid A \to \alpha \bullet X^*\beta\rangle$	$\Rightarrow$	$\langle \mathbf{S} \mid A \to \alpha \bullet \beta\rangle$	

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 $\star$ qualifiers we add the starred inference rules where COMPLETE* and GENERATE* implement an iteration that stops with the CLOSE* rule.

Generator State (stack)	**Action**
$\langle \texttt{expr} \to \bullet\texttt{expr}^*\rangle$	START(List)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \bullet\texttt{expr operator expr}\rangle$	PREDICT*(BinOp)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \bullet\texttt{expr operator expr} \mid \texttt{expr} \to \bullet\texttt{constant}\rangle$	PREDICT(Constant)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \bullet\texttt{expr operator expr} \mid \texttt{expr} \to \texttt{constant}\bullet\rangle$	GENERATE(7)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr} \bullet \texttt{operator expr}\rangle$	COMPLETE
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr} \bullet \texttt{operator expr} \mid \texttt{expr} \to \bullet\rangle$	PREDICT(Add)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr operator} \bullet \texttt{expr}\rangle$	COMPLETE
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr operator} \bullet \texttt{expr} \mid \texttt{expr} \to \bullet\texttt{constant}\rangle$	PREDICT(Constant)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr operator} \bullet \texttt{expr} \mid \texttt{expr} \to \texttt{constant}\bullet\rangle$	GENERATE(5)
$\langle \texttt{expr} \to \bullet\texttt{expr}^* \mid \texttt{expr} \to \texttt{expr operator expr}\bullet\rangle$	COMPLETE
$\langle \texttt{expr} \to \texttt{expr} \bullet \texttt{expr}^*\rangle$	COMPLETE*
$\langle \texttt{expr} \to \texttt{expr} \bullet \texttt{expr}^* \mid \texttt{expr} \to \bullet\texttt{constant}\rangle$	PREDICT*(Constant)
$\langle \texttt{expr} \to \texttt{expr} \bullet \texttt{expr}^* \mid \texttt{expr} \to \texttt{constant}\bullet\rangle$	GENERATE(4)
$\langle \texttt{expr} \to \texttt{expr expr} \bullet \texttt{expr}^*\rangle$	COMPLETE*
$\langle \texttt{expr} \to \texttt{expr expr}\bullet\rangle$	CLOSE*

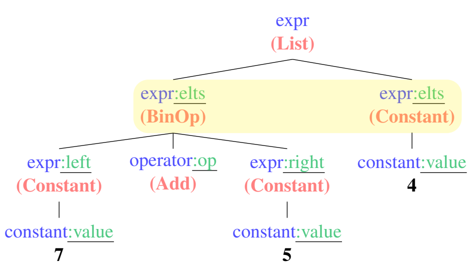

Figure 3: Example derivation for the generation of the Python list expression `[7+5,4]`. The derivation starts with `expr` as axiom symbol and applies transitions until the goal is reached. The list of actions performed is called the generator **derivation**. Given a generated derivation we can design 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:

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

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 hypothesis among all candidate hypotheses by solving the following optimization problem:

$$\hat{y} = \operatorname*{argmax}_y p(y|x) \tag{2}$$

The family of models varies according to four key qualitative factors that we identify in the TranX architecture. First we describe a substitution procedure managing variables and lists names in section 3.1). Second, in section 3.2, we test the architectural variations for encoding the natural language sequence. Third, in section 3.3, we describe variations related to constraining the generated code with grammatical constraints and architectural variations 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 predefined 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:

> *create list `` `done` `` containing permutations of each element in list `` `[a, b, c, d]` `` with variable `` `x` `` as tuples*

is transformed into:

> *create list `var_0` containing permutations of each element in list `lst_0` with variable `var_1` as tuples*

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

Models using variable replacement as illustrated above, are identified with the notation SUBSTITUTION = 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 $\mathrm{BERT}_{\mathrm{BASE}}$ to encode the input natural language $\{x_i, i \in [\![1, n]\!]\}$ of n words into a vectorial representations $\{h_i^{(\mathrm{enc})}, i \in [\![1, n]\!]\}$ 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 $h_t^{(\mathrm{dec})}$:

$$h_t^{(\mathrm{dec})} = \mathrm{LSTM}([e_{t-1} : \tilde{a}_{t-1}], h_{t-1}^{(\mathrm{dec})}) \tag{3}$$

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 Luong et al. (2015) combining the weighted average over all the source hidden state c_t and the decoder hidden state $h_t^{(\mathrm{dec})}$:

$$\tilde{a}_t = W_a[c_t : h_t^{(\mathrm{dec})}] \tag{4}$$

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:

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

These models are not constrained by the Python grammar and we identify these models with GRAMMAR = FALSE.

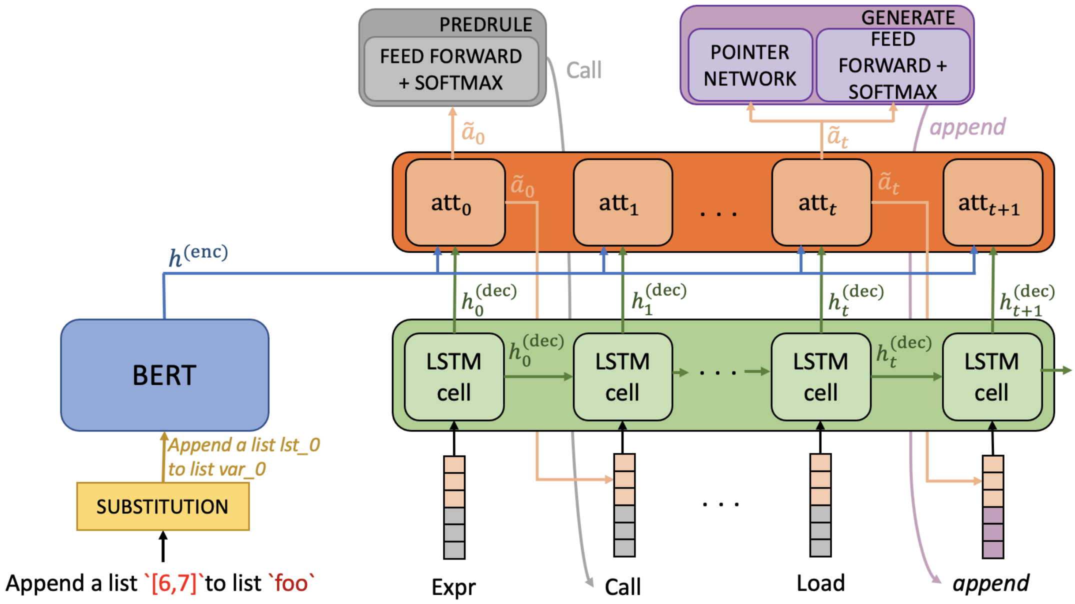

Figure 4: Illustration of the seq2seq model with the variables SUBSTITUTION, GRAMMAR, BERT, POINTERNET set to TRUE. We describe here the complete process where we predict a derivation sequence composed of grammar rules and CLOSE (PREDRULE) or Python variables/built-in (GENERATE). The astor library is used to transform the AST constructed with the derivation sequence into Pyton code. In the case where GRAMMAR = FALSE, we only have the GENERATE action which exclusively predicts unconstrained code tokens (as for a classical seq2seq).

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:

$$\begin{aligned} p(y_t = \mathrm{GENERATE}[v]|x, e_{<t}) = \\ p(\mathrm{gen}|x, e_{<t})p(v|\mathrm{gen}, x, e_{<t}) \\ +p(\mathrm{copy}|x, e_{<t})p(v|\mathrm{copy}, x, e_{<t}) \end{aligned} \quad (6)$$

The probabilities $p(\mathrm{gen}|.)$ and $p(\mathrm{copy}|.)$ sum to 1 and are computed with $\mathrm{softmax}(W \cdot \tilde{a}_t)$. The probability of generating v from the vocabulary V $p(v|\mathrm{gen}, .)$ is defined in the same way as (5). We use the pointer net architecture (Vinyals et al., 2015) to compute the probability $p(v|\mathrm{copy}, .)$ of copying an element from the natural language x. Models that use a pointer network are identified with PN = TRUE, otherwise with PN = 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 nonterminal symbol, the PREDRULE is used. This task either outputs a PREDICT(C) action or the CLOSE action:

$$\begin{aligned} p(y_t = \mathrm{PREDRULE}[c]|x, e_{<t}) = \\ \mathrm{softmax}(e_r^\top \cdot W_p \cdot \tilde{a}_t) \end{aligned} \quad (7)$$

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 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 GRAMMAR = TRUE. Recall that the 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* $\rightarrow$ `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 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 $\alpha = 5.10^{-3}$.

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 $\alpha = 5.10^{-5}$. 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

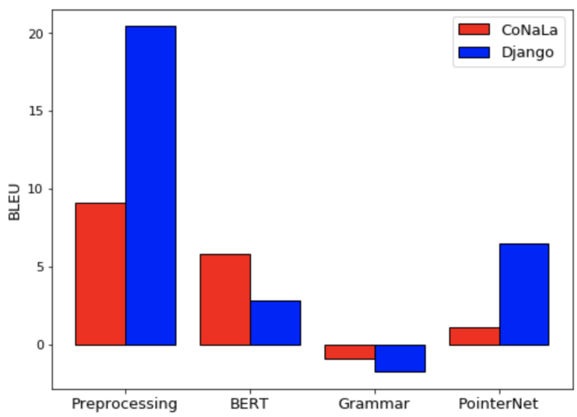

Figure 5: Difference between the marginal mean of each variable for the TRUE and FALSE conditions.

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.

Substitution	BERT	Grammar	PN	CoNaLa BLEU	CoNaLa accuracy	Django BLEU	Django accuracy
False	False	False	False	21.05 ± 0.81	0.9 ± 0.42	42.58 ± 1.54	26.86 ± 1.15
			True	22.33 ± 0.78	1.7 ± 0.90	64.79 ± 1.00	62.85 ± 1.21
		True	False	20.59 ± 0.74	2.87 ± 0.48	43.23 ± 1.62	30.12 ± 0.63
			True	22.16 ± 1.93	3.87 ± 1.65	62.55 ± 1.60	65.20 ± 0.03
	True	False	False	30.83 ± 4.08	2 ± 0.94	53.18 ± 0.87	30.28 ± 0.26
			True	30.98 ± 1.33	3.3 ± 1.48	58.69 ± 1.28	37.96 ± 0.27
		True	False	25.88 ± 0.94	3.8 ± 1.96	47.32 ± 0.50	29.62 ± 0.33
			True	28.43 ± 0.64	4.4 ± 1.67	52.55 ± 0.51	37.38 ± 0.38
True	False	False	False	31.17 ± 0.88	3.1 ± 1.52	70.4 ± 0.25	70.40 ± 0.29
			True	32.10 ± 1.06	3.1 ± 1.24	70.28 ± 0.38	70.46 ± 0.37
		True	False	33.36 ± 1.63	6.37 ± 0.63	70.82 ± 0.22	71.3 ± 0.19
			True	32.86 ± 1.75	5 ± 1.67	70.62 ± 0.49	71.47 ± 0.19
	True	False	False	36.43 ± 0.41	4.5 ± 1.84	$\mathbf{76.97 \pm 0.15}$	74.58 ± 0.27
			True	36.29 ± 2.27	5 ± 1.32	76.62 ± 0.50	$\mathbf{76 \pm 0.71}$
				35.42 ± 1.75*	5.2 ± 1.33*	-	-
		True	False	35.04 ± 1.03	7.3 ± 1.25	76.20 ± 0.46	74.88 ± 0.56
			True	37.99 ± 1.85	7.5 ± 1.12	76.32 ± 0.59	75.32 ± 1.54
				$\mathbf{39.01 \pm 1.08}$*	$\mathbf{7.7 \pm 1.92}$*	-	-

Table 1: Performances with different natural language encoders on the development sets with and without a grammatical component. The scores reported are the mean and standard deviation resulting from training with 5 different seeds. The * refers to the use of 100k CoNaLa mined data in addition to clean examples.

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 code should be `1.append([6,7])`. However the AST library we used prevents from generating such ill-typed code. The absence of code generation in such cases explain the decrease in BLEU score.

The use of substitution partially corrects for these typing errors because the substituted symbols `var_i, lst_i` are generally more likely to be predicted and are likely to have the right type thanks to the mapping.

Grammatical aspect The transition system doesn't improve the results on average because

System	CoNaLa BLEU	CoNaLa accuracy	Django BLEU	Django accuracy
(Yin and Neubig, 2018)	27.2	-	-	73.7
(Yin and Neubig, 2018) + mined	28.1	-	-	-
(Orlanski and Gittens, 2021) + mined 100k	30.55	-	-	-
(Norouzi et al., 2021) + 600k mined	32.57	-	-	**81.03**
Ours BERT + GRAMMAR	31.6	4.5	79.86	79.77
Ours BERT + GRAMMAR + 100k mined	**34.20**	**5.8**	-	-
Ours BERT (tokens)	30.73	1.40	79.81	79.61
Ours BERT + 100k mined (tokens)	32.39	3.4	-	-

Table 2: Comparisons of the systems trained without external data sources on CoNaLa and Django test sets.

of the empty predictions when SUBSTITUTION = FALSE. The use of the transition system leads to better results when SUBSTITUTION = TRUE but not as drastically as one would have expected. However the real contribution of the grammar associated with substitution is the syntactic validity of the code in 100% of the cases, as tested with our architecture obtaining the best results. In scenarios where we do not use the grammar, it is never the case to have an empty output. But then the proportion of code sequences that are actually syntactically valid in this setup is 92% on average.

BERT As expected when using BERT to encode the natural language input we get an improvement of about 6 marginal BLEU on CoNaLa (respectively +3 BLEU on Django). More interestingly, this effect is lower than the one of the substitution operation.

We conclude that the use of a pre-trained model increases the results but less than substitution, despite what one might think and it suggests that improving the management of variable names and lists is one of the key elements for improving the system. The contribution of grammatical constraints in BLEU may seem detrimental but we could see that this is a side effect of typing constraints in adversarial scenarios. Overall the non-constrained generated code is syntactically incorrect in 8% of the cases.

4.5 Test

We compare in table 2 our results with other systems on CoNaLa and Django test sets. We report our best performing models on the development set with and without grammatical constraints. We also use models trained on the full CoNaLa including mined examples to get relevant comparisons.

Among the other systems Yin and Neubig (2018) is the only one that uses grammatical constraints. Our architecture differs with the use of a BERT encoder whereas Yin and Neubig (2018) use an LSTM. The other systems do not use grammatical constraints but rather try to take advantage of additional data. Orlanski and Gittens (2021) and Norouzi et al. (2021) aim to take advantage of the CoNaLa mined examples. As these mined examples are noisy, Orlanski and Gittens (2021) takes advantage of BART (Lewis et al., 2020), a denoising encoder. They also enrich the natural language input with the results of queries from StackOverflow by adding the title of the post, its associated tags, etc. Norouzi et al. (2021) use BERT as encoder and a transformer decoder. They apply the Target Autoencoding method introduced by Currey et al. (2017). During training, the encoder parameters are frozen and the decoder is trained to reconstruct code examples. They use this method on the mined examples to take maximal advantage of the additional noisy data.

We observe that our grammar based model with BERT encoder is state of the art on CoNaLa while the transformer encoder/decoder architecture of Norouzi et al. (2021) performs best on Django. Quite interestingly the exact match accurracy of these models remain weak on CoNaLa.

5 Conclusion

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

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. CoRR, abs/2108.07732.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. volume abs/2107.03374.

Anna Currey, Antonio Valerio Miceli Barone, and Kenneth Heafield. 2017. Copied monolingual data improves low-resource neural machine translation. In Proceedings of the Second Conference on Machine Translation, WMT 2017, Copenhagen, Denmark, September 7-8, 2017, pages 148–156. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. pages 4171–4186.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics.

Jay Earley. 1970. An efficient context-free parsing algorithm. Commun. ACM, 13(2):94–102.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. CoRR, abs/2105.09938.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, pages 7871–7880. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. pages 1412–1421.

Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code generation from natural language with less prior knowledge and more monolingual data. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021, pages 776–785. Association for Computational Linguistics.

Gabriel Orlanski and Alex Gittens. 2021. Reading stackoverflow encourages cheating: Adding question text improves extractive code generation. CoRR, abs/2106.04447.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pages 1139–1149. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. pages 2692–2700.

Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. 1997. The zephyr abstract syntax description language. In Proceedings of the Conference on Domain-Specific Languages, October 15-17, 1997, Santa Barbara, California, USA, pages 213–228.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pages 440–450. Association for Computational Linguistics.

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

A Additional Qualitative Examples

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

Source	*convert tuple ‘t‘ to list*
Gold	`list(t)`
Grammar	`[x for x in t for x in t]`
Without Grammar	`[i for i in t]`
Remark	Problem of CLOSE for the Grammar output. Without grammar the code is correct but with a low BLEU.

Source	*get the position of item 1 in ‘testlist‘*
Gold	`[i for i, x in enumerate(testlist) if x == 1]`
Grammar	`[i for i, v in enumerate(testlist) if v == 1]`
Without Grammar	`testlist = [i for i in testlist if i != 1]`
Remark	Grammar output is not equal to Gold due to dummy variable.

Source	*declare an array*
Gold	`my_list = []`
Grammar	`x = [0] * 2`
Without Grammar	`[(0) for _ in range (10000)]`
Remark	Source is not precise enough. Models’ outputs are valid.

Source	*append a numpy array ‘b‘ to a numpy array ‘a‘*
Gold	`np.vstack((a, b))`
Grammar	`a = numpy.array([b, a])`
Without Grammar	`z = np.array([b]).reshape((3, 3))`
Remark	Gold is not accurate with `np` undefined before. `vstack` function not in the external vocabulary.

Source	*activate is a lambda function which returns None for any argument x.*
Gold	`activate = lambda x : None`
Grammar	`activate = lambda x = None : x`
Without Grammar	`activate = lambda x : None`
Remark	Good BLEU for grammar output while the result is not adequate.

Source	*remove first element of text*
Gold	`text = text[1:]`
Grammar	`text = text[1:]`
Without Grammar	`text[1:`
Remark	Syntax mistake for the code without grammar.

Source	*increment piece by first element of elt*
Gold	`piece += elt[0]`
Grammar	`piece += elt[1]`
Without Grammar	`piece += elt[1]`
Remark	First element of a list is zero, not one.