Code2Text: Dual Attention Syntax Annotation Networks for Structure-Aware Code Translation

Yun Xiong[1(✉)], Shaofeng Xu[1], Keyao Rong[1], Xinyue Liu[2], Xiangnan Kong[2], Shanshan Li[3], Philip Yu[4], and Yangyong Zhu[1]

[1] Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China
{yunx,sfxu16,18212010024,yyzhu}@fudan.edu.cn

[2] Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, USA
{xliu4,xkong}@wpi.edu

[3] School of Computer Science, National University of Defense Technology, Changsha, China
shanshanli@nudt.edu.cn

[4] University of Illinois at Chicago, Chicago, IL, USA
psyu@uic.edu

Abstract. Translating source code into natural language text helps people understand the computer program better and faster. Previous code translation methods mainly exploit human specified syntax rules. Since handcrafted syntax rules are expensive to obtain and not always available, a PL-independent automatic code translation method is much more desired. However, existing sequence translation methods generally regard source text as a plain sequence, which is not competent to capture the rich hierarchical characteristics inherently reside in the code. In this work, we exploit the abstract syntax tree (AST) that summarizes the hierarchical information of a code snippet to build a structure-aware code translation method. We propose a syntax annotation network called Code2Text to incorporate both source code and its AST into the translation. Our Code2Text features the dual encoders for the sequential input (code) and the structural input (AST) respectively. We also propose a novel dual-attention mechanism to guide the decoding process by accurately aligning the output words with both the tokens in the source code and the nodes in the AST. Experiments on a public collection of Python code demonstrate that Code2Text achieves better performance compared to several state-of-the-art methods, and the generation of Code2Text is accurate and human-readable.

Keywords: Natural language generation · Tree-LSTM · Abstract syntax tree · Data mining

1 Introduction

We have witnessed a large amount of source code been released in recent years, manually writing detailed annotations (*e.g.*, comments, pseudocode) for them is a tedious and time-consuming task. However, these annotations play an irreplaceable role in the development of software. For example, it serves as a guideline for the new engineers to quickly understand the functionality of each piece of code, and it helps one grasp the idea of legacy code written in a less popular programming language. Accordingly, an effective automatic source code translation method is desired, where the goal is to translate the code into a corresponding high-quality natural language translation (we also refer the translated text as *annotation* for short throughout this paper).

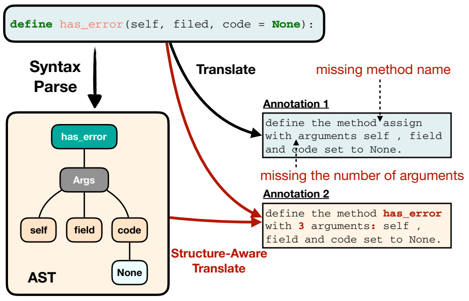
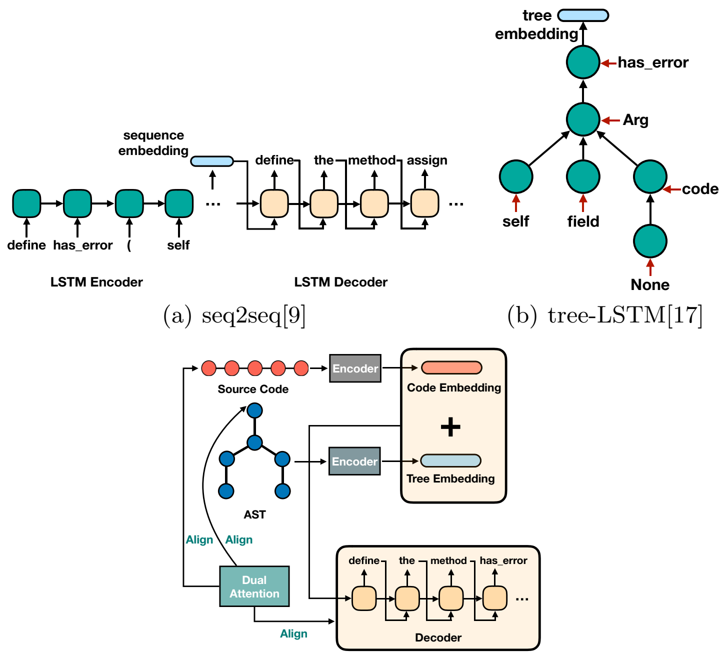

define has_error(self, filed, code = None):
Syntax Parse
Translate
missing method name
Annotation 1
define the method assign with arguments self , field and code set to None.
missing the number of arguments
Annotation 2
define the method has_error with 3 arguments: self , field and code set to None.
has_error
Args
self
field
code
None
AST
Structure-Aware Translate

Fig. 1. An illustration of the syntax structure-aware code translation task. We propose to exploit the abstract syntax tree (AST) that reflects the syntax structure of the code snippet for an improved code translation (Annotation 2). While conventional sequence translation methods can only use the sequential input (the code), which may miss key structural information in the code syntax (Annotation 1). (Color figure online)

Given the significance of the code translation problem, most of the existing methods [1,19] only follow the common natural language translation routine by treating the code as a plain sequence. However, programming languages obey much more strict syntax rules than natural languages [8], which should not be ignored in translation. Given the syntax rules of most programming languages are explicitly defined, each piece of legal code could be represented by a structural representation called abstract syntax tree (AST). In Fig. 1, the AST of a simple function declaration (blue box) in Python is illustrated in the underneath yellow box, which depicts the syntax structure of the code. The plain sequence translator who only takes the code as input may have difficulty in capturing the structural information, so the generated annotation may omit crucial details reside in the original code, such as the function name and the number of parameters. We show that this limitation could be fixed by considering the corresponding AST, as it represents the code structure and the hierarchical relations that are hard to learn sequentially.



(a) seq2seq[9] (b) tree-LSTM[17]

(c) Code2Text(this paper)

Fig. 2. Comparison of related methods. (a) Seq2seq model consists of an LSTM encoder and an LSTM decoder. (b) Tree structured recurrent neural network which encodes structured text from children to parent nodes. (c) Our Code2Text encodes both sequential and hierarchical information to generate the target sequence. Details on the dual-encoder and dual-attention are illustrated in Fig. 3.

Thus, it is interesting to investigate whether and how one could combine the information in source code and its AST for an improved structure-aware code translation model. Although ASTs are much easier to obtain compared to handcrafted syntax rules, it is not a trivial task to incorporate it into the translation method due to the following reasons.

Sequential and Hierarchical Encoding: The encoder plays a vital role in correctly understanding the semantics of the input text in code translation. Existing methods usually employ RNN/LSTM to learn the dependencies between words according to their sequential orders, as shown in Fig. 2(a). However, it is not applicable to hierarchical inputs such as AST. To additionally consider the hierarchical patterns of the input source code, we need an approach to encode the tree-structured inputs. Besides, we also need the sequential encoder to process the source code since ASTs focus more on structure rather than token-level details. How to effectively combine the sequential encoder and hierarchical encoder is also an open question.

Tree to Sequence Alignment in Hierarchical Attention: Attention mechanism is an important component in machine translation. It helps the decoder choose more reasonable and accurate tokens by aligning generated words with the words from the input at each step. Without attention, the decoder may generate redundant words or miss some words from the source text. As shown in Fig. 2(c), to obtain the best annotation, we need to align the decoding step with the words in the source code and the nodes in AST simultaneously. However, the existing attention models only work on sequential data. It is unknown and challenging to align each decoding step with the nodes in the AST.

To tackle the challenges above, we propose a novel model called Code2Text. As Fig. 2(c) and Fig. 3 show, our Code2Text informatively incorporates hierarchical information from code AST, and trains a dual-encoder sequence to sequence language model with improved attention mechanism for word alignment. Experiments on the open-source Python project dataset reveal that our model achieves better performance than state-of-the-art algorithms. Several case studies are displayed to demonstrate that Code2Text generates accurate and understandable annotations as we pursued.

We summarize our contributions as follow,

1. We are the first work to incorporate code structure information into a code annotation task.
2. We create a model, Code2Text, with a dual-encoder which can encode both semantic information from source code and hierarchical information of the AST. We also propose a dual-attention mechanism to improve the original attention mechanism by extending it to align the structural AST tokens.
3. We perform extensive experiments on a public benchmark Python code dataset. Other than the numerical evaluation, we additionally present case studies to demonstrate the effectiveness of our dual-attention encoder design.

2 Preliminary

2.1 Problem Definition

Let's take a look at the definition of NMT (Neural Machine Translation) first. Suppose we have a dataset $\mathcal{D} = \{(\mathbf{x}^s, \mathbf{y})\}$ and the corresponding annotation. $\mathcal{X}^s$ and $\mathcal{Y}$ are sets of source code and annotation, respectively. $\mathbf{x}^s = (x_1^s, \cdots, x_n^s)$ represents a sequence of source code with n words and $\mathbf{y} = (y_1, \cdots, y_m)$ represents a sequence of annotation with m words. Our task is to translate $\mathbf{x}^s$ to $\mathbf{y}$ for each pair in dataset $\mathcal{D}$, which is the same task as translating a source language to a target language. The overall goal of normal Neural Machine Translation models is to estimate the conditional probability distribution $\Pr(\mathcal{Y}|\mathcal{X}^s)$. Conventional inference approaches usually require i.i.d. assumptions, and ignore dependency between different instances. The inference for each instance is performed independently:

$$\Pr(\mathcal{Y}|\mathcal{X}^s) \propto \prod_{(\mathbf{x}^s,\mathbf{y})\in\mathcal{D}} \Pr(\mathbf{y}|\mathbf{x}^s) \tag{1}$$

In this work, our model considers not only semantic information but also hierarchical information. Therefore, we derive another symbol $\mathbf{x}^t = (x_1^t, \cdots, x_q^t)$, who contains AST information of relative source code $\mathbf{x}^s$ with q words. $\mathcal{X}^t$ is a set which contains $\mathbf{x}^t$. Accordingly, we create an extended dataset $\mathcal{D}' = \{(\mathbf{x}^s, \mathbf{x}^t, \mathbf{y})\}$. To incorporate hierarchical information, we will modify our probability distribution:

$$\Pr(\mathcal{Y}|\mathcal{X}^s, \mathcal{X}^t) \propto \prod_{(\mathbf{x}^s, \mathbf{x}^t, \mathbf{y}) \in \mathcal{D}'} \Pr(\mathbf{y}|\mathbf{x}^s, \mathbf{x}^t) \tag{2}$$

2.2 Attention Seq2seq Model

Attention seq2seq model is a sophisticated end-to-end neural translation approach, which consists of the encoder process and decoder process with an attention mechanism.

Encoder. In the encoder process, we aim to embed a sequence of source code $\mathbf{x}^s = (x_1^s, \cdots, x_n^s)$ into d-dimension vector space.

We usually replace vanilla RNN [10] unit with LSTM (Long Short Term Memory) [4] unit due to the gradient explosion/vanishing problem. The j-th LSTM unit has three *gates*: an input gate $\mathbf{i}_j^s \in \mathcal{R}^{d\times 1}$, a forget gate $\mathbf{f}_j^s \in \mathcal{R}^{d\times 1}$ and an output gate $\mathbf{o}_j^s \in \mathcal{R}^{d\times 1}$ and two states: a hidden state $\mathbf{h}_j^s \in \mathcal{R}^{d\times 1}$ and a memory cell $\mathbf{c}_j^s \in \mathcal{R}^{d\times 1}$. Update rules for an LSTM unit are below:

$$\mathbf{i}_j^s = \sigma(\mathbf{W}^{(i)} embed(x_j^s) + \mathbf{U}^{(i)} \mathbf{h}_{j-1}^s + \mathbf{b}^{(i)}), \tag{3}$$

$$\mathbf{f}_j^s = \sigma(\mathbf{W}^{(f)} embed(x_j^s) + \mathbf{U}^{(f)} \mathbf{h}_{j-1}^s + \mathbf{b}^{(f)}), \tag{4}$$

$$\mathbf{o}_j^s = \sigma(\mathbf{W}^{(o)} embed(x_j^s) + \mathbf{U}^{(o)} \mathbf{h}_{j-1}^s + \mathbf{b}^{(o)}), \tag{5}$$

$$\tilde{\mathbf{c}}_j^s = \tanh(\mathbf{W}^{(\tilde{c})} embed(x_j^s) + \mathbf{U}^{(\tilde{c})} \mathbf{h}_{j-1}^s + \mathbf{b}^{(\tilde{c})}), \tag{6}$$

$$\mathbf{c}_j^s = \mathbf{i}_j^s \odot \tilde{\mathbf{c}}_j^s + \mathbf{f}_j^s \odot \mathbf{c}_{j-1}^s, \tag{7}$$

$$\mathbf{h}_j^s = \mathbf{o}_j^s \odot \tanh(\mathbf{c}_j^s), \tag{8}$$

Here, $\tilde{\mathbf{c}}_j^s \in \mathcal{R}^{d\times 1}$ denotes the state for updating the memory cell $\mathbf{c}_j^s$. Function *embed*() turns a word into a d-dimension embedding vector. It can be assigned with a fixed global word vector or trained by the model itself. $\mathbf{W}^{(\cdot)}, \mathbf{U}^{(\cdot)} \in \mathcal{R}^{d\times d}$ are weight matrix and $\mathbf{b}^{(\cdot)} \in \mathcal{R}^{d\times 1}$ is a bias vector. σ is the logistic function and the operator $\odot$ means element-wise product between two vectors. We initialize $\mathbf{h}_0^s$ as a d-dimension vector of all zeros, and iterate over the sequence and finally obtain $\mathbf{h}_n^s$ at the end of the source sentence, which represents the information of source code.

Decoder. After we obtain source code representation vector $\mathbf{h}_n^s$ from the encoder process, we then predict the annotation sequence with LSTM in a similar way in the decoder process. We define $\mathbf{d}_j$ as the j-th hidden state. Given the input embedding vector $embed(\mathbf{x}^s)$ and previous word sequence $\mathbf{y}_{<j}$, we generate j-th word by estimating the conditional probability:

$$p(y_j|\mathbf{y}_{<j}, embed(\mathbf{x}^s)) = softmax(\mathbf{d}_j), \tag{9}$$

where $softmax()$ function produces probabilities according to the j-th hidden state $\mathbf{d}_j$, and $\mathbf{d}_j$ is calculated by another non-linear function f_d as follows:

$$\mathbf{d}_j = f_d(\mathbf{d}_{j-1}, embed(y_{j-1})), \tag{10}$$

We initialize $\mathbf{d}_0 = \mathbf{h}_n^s$ to ensure that our predictor can generate an annotation sequence base on source code sequential information.

Attention Mechanism. Attention mechanism [9] was proposed to align each decoder hidden state with the encoder output states. With the attention process, we can explicitly calculate the contribution each encoder output state made to the word prediction at each step.

Suppose we have hidden state $\mathbf{d}_j$ at time j in the decoder process, and $(\mathbf{h}_1^s, \cdots, \mathbf{h}_n^s)$ are encoder hidden states. According to [9], we first calculate attention weights α_{ij}^s between the i-th hidden state $\mathbf{h}_i^s$ in encoder and the j-th hidden state $\mathbf{d}_j$ in decoder as follows:

$$\alpha_{ij}^s = \frac{\exp(score(\mathbf{h}_i^s, \mathbf{d}_j))}{\sum_{k=1}^{n} \exp(score(\mathbf{h}_k^s, \mathbf{d}_j))}, \tag{11}$$

where $score()$ function is used to compare the decoder hidden state $\mathbf{d}_j$ with each of the source hidden states $\mathbf{h}_i^s$, and the result is normalized to produce attention weights (a distribution over source positions). Then based on attention weights we compute j-th context vector $\mathbf{w}_j^s$ as the weighted average of the source encoder hidden states:

$$\mathbf{w}_j^s = \sum_{i=1}^{n} \alpha_{ij}^s \mathbf{h}_i^s, \tag{12}$$

Afterward, we apply a non-linear function *tanh* to the concatenation of the context vector $\mathbf{w}_j^s$ and the current decoder hidden state $\mathbf{d}_j$, and yield the final attention vector $\mathbf{a}_j$:

$$\mathbf{a}_j = \tanh(\mathbf{W}_d \times (\mathbf{d}_j \oplus \mathbf{w}_j^s) + \mathbf{b}_d), \tag{13}$$

where $\oplus$ means concatenation of $\mathbf{d}_j$ and $\mathbf{w}_j^s$. $\mathbf{W}_d \in \mathcal{R}^{d \times 2d}$ is a weight matrix and $\mathbf{b}_d \in \mathcal{R}^{d \times 1}$ is a bias vector. Once computed, the attention vector $\mathbf{a}_j$ is used to derive the softmax logit:

$$p(y_j|\mathbf{y}_{<j}, \mathbf{x}^s) = softmax(\mathbf{a}_j). \tag{14}$$

In Sect. 4 our experiments will present performance and cases of this seq2seq model.

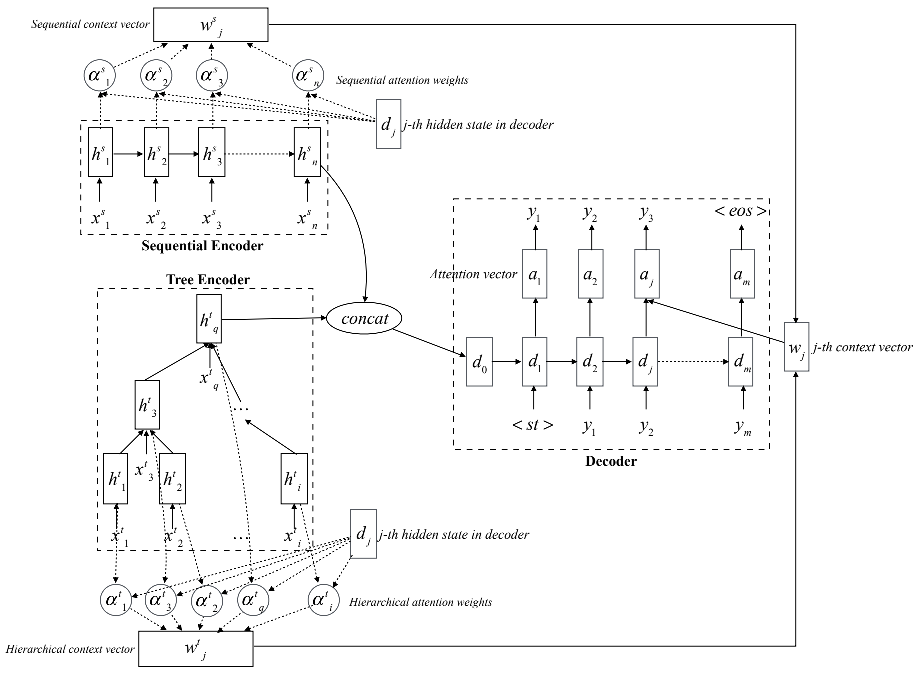

Sequential context vector
w^s_j
Sequential attention weights
d_j j-th hidden state in decoder
Sequential Encoder
Tree Encoder
concat
Attention vector
j-th context vector
Decoder
Hierarchical attention weights
Hierarchical context vector
w^t_j

Fig. 3. Architecture of our Code2Text model. $(x_1, x_2, \cdots, x_n)$ are tokens of source code, $(t_1, t_2, \cdots, t_q)$ are tokens of AST and $(y_1, y_2, \cdots, y_m)$ are tokens of natural language annotation. $< st >$ and $< eos >$ are start and end tokens, respectively. Encoder process consists of sequence encoder and tree encoder, and $\mathbf{h}_0$ is initialized with a vector of all zeros. In the decoder process, $\mathbf{d}_0$ is initialized with the bilinear result of $\mathbf{h}_n$ and $\mathbf{h}^t_q$. Context vector $\mathbf{c}$ in attention mechanism is the concatenation of sequence context vector and hierarchical context vector.

3 Proposed Method: Code2Text

In this section, we will formally introduce our model, Code2Text, which is an extension and improvement of original seq2seq models. We first propose a dual-encoder by creating a tree encoder for representing the summary of AST information along with the original sequential encoder in the encoder process, then we explain how our dual-attention mechanism works by incorporating hierarchical outputs from tree encoder. The architecture of our model shows in Fig. 3.

3.1 Dual-Encoder

As Fig. 3 describes, in our encoder process, the sequential encoder produces sequential representation $\mathbf{h}^s_n$, which will be a part of our dual-encoder information. The other part, *tree encoder*, will produce hierarchical representation from AST of source code.

Sequential Encoder. The encoder introduced in Sect. 2.2 would be employed as our sequential encoder directly.

Tree Encoder. Now we formally formulate our tree encoder. For each pair of source code sequence $\mathbf{x}^s = (x_1^s, x_2^s, \cdots, x_n^s)$ and annotation words sequence $\mathbf{y} = (y_1, y_2, \cdots, y_m)$, we preprocess by parsing $\mathbf{x}^s$ to AST sequence $\mathbf{x}^t = (x_1^t, x_2^t, \cdots, x_q^t)$ and AST parent index list $\mathbf{p} = (p_1, p_2, \cdots, p_q)$. Here, q is equal to the number of words in AST sequence.

Our tree encoder aims to represent AST with a vector, hence, for propagating information from children nodes to the root node, we employ a special LSTM unit, *tree-LSTM* [17] to our tree encoder. Tree-LSTM was proposed to improve semantic representations on tree-structured network topologies, which is appropriate for our work. There are two architectures: the *Child-Sum Tree-LSTM* and the *N-ary Tree-LSTM*. Code AST is a natural kind of dependency trees, and Child-Sum Tree-LSTM is a good choice for dependency trees [17]. However, N-ary Tree-LSTMs are suited for constituency trees which are not suitable for AST in our task. Therefore, we will choose Child-Sum Tree-LSTM in our work. We denote $\mathcal{C}_j$ as the children of j-th node in a AST. The hidden state $\mathbf{h}_j^t \in \mathcal{R}^{d\times 1}$ and memory cell $\mathbf{c}_j^t \in \mathcal{R}^{d\times 1}$ for j-th node are updated as follows:

$$\tilde{\mathbf{h}}_j^t = \sum_{k\in\mathcal{C}(j)} \mathbf{h}_k^t, \tag{15}$$

$$\mathbf{i}_j^t = \sigma(\mathbf{W}^{(i^t)} embed(x_j^t) + \mathbf{U}^{(i^t)}\tilde{\mathbf{h}}_j^t + \mathbf{b}^{(i^t)}), \tag{16}$$

$$\mathbf{f}_{jk}^t = \sigma(\mathbf{W}^{(f^t)} embed(x_j^t) + \mathbf{U}^{(f^t)}\mathbf{h}_k^t + \mathbf{b}^{(f^t)}), \tag{17}$$

$$\mathbf{o}_j^t = \sigma(\mathbf{W}^{(o^t)} embed(x_j^t) + \mathbf{U}^{(o^t)}\tilde{\mathbf{h}}_j^t + \mathbf{b}^{(o^t)}), \tag{18}$$

$$\tilde{\mathbf{c}}_j^t = \tanh(\mathbf{W}^{(\tilde{c}^t)} embed(x_j^t) + \mathbf{U}^{(\tilde{c}^t)}\tilde{\mathbf{h}}_j^t + \mathbf{b}^{(\tilde{c}^t)}), \tag{19}$$

$$\mathbf{c}_j^t = \mathbf{i}_j^t \odot \tilde{\mathbf{c}}_j^t + \sum_{k\in\mathcal{C}(j)} \mathbf{f}_{jk}^t \odot \mathbf{c}_k^t, \tag{20}$$

$$\mathbf{h}_j^t = \mathbf{o}_j^t \odot \tanh(\mathbf{c}_j), \tag{21}$$

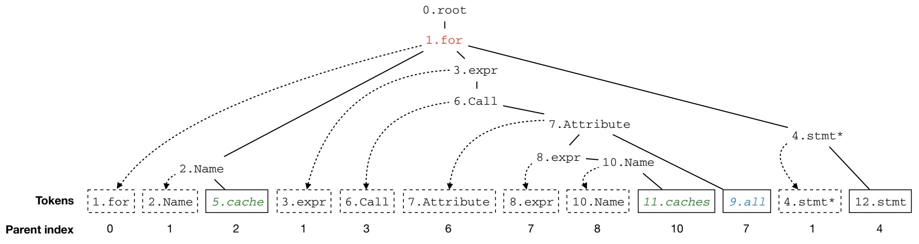


Fig. 4. For each AST, we extract node tokens and put them into a token array, then assign parent index for each token.

where $k \in \mathcal{C}_j$ in Eq. 17, $\tilde{\mathbf{h}}_j^t \in \mathcal{R}^{d\times 1}$ is the sum of children hidden state, $\tilde{\mathbf{c}}_j^t \in \mathcal{R}^{d\times 1}$ denotes the state for updating the memory cell $\mathbf{c}_j^t$. $\mathbf{i}_j^t, \mathbf{o}_j^t, \mathbf{f}_{jk}^t \in \mathcal{R}^{d\times 1}$ are input gate, output gate and forget gate, respectively. $\mathbf{W}^{(\cdot)}, \mathbf{U}^{(\cdot)} \in \mathcal{R}^{d\times d}$ are weight matrix and $\mathbf{b}^{(\cdot)} \in \mathcal{R}^{d\times 1}$ is a bias vector. σ is the logistic function and the operator $\odot$ means element-wise product between two vectors.

3.2 Decoder

From the encoder process, we obtain two embedded vectors, sequential representation vector $\mathbf{h}_n^s$ and tree representation vector $\mathbf{h}_q^t$. Afterward, we initialize decoder hidden state $\mathbf{d}_0$ with the concatenation of $\mathbf{h}_n^s$ and $\mathbf{h}_q^t$ along the sequence length dimension:

$$\mathbf{d}_0 = \mathbf{h}_n^s \oplus \mathbf{h}_q^t, \tag{22}$$

where $\oplus$ means concatenation operation. This decoder initialization considers not only source code sequential summary but also AST structure summary, which could improve predictor performance than the original seq2seq model. The rest decoder process remains the same.

3.3 Dual-Attention Mechanism

After introducing our dual-encoder, we need to improve the attention mechanism to adopt hierarchical hidden outputs from tree encoder. The main difference between our dual-attention and original attention mentioned in Sect. 2.2 is the construction of context vector $\mathbf{w}^t$ for tree encoder. Concretely, as Fig. 3 shows, at j-th step, α_{ij}^s in attention seq2seq still represents sequential attention weights, and hierarchical attention weights α_{ij}^t are calculated by treating them the same as sequential outputs:

$$\alpha_{ij}^t = \frac{\exp(score(\mathbf{h}_i^t, \mathbf{d}_j))}{\sum_{k=1}^q \exp(score(\mathbf{h}_k^t, \mathbf{d}_j))}, \tag{23}$$

As in [2], we parameterize the score function $score()$ as a feed-forward neural network which is jointly trained with all the other components of the proposed architecture. Then we compute j-th context vector $\mathbf{w}_j$ as the weighted average of the sequential hidden states and tree hidden states:

$$\mathbf{w}_j = \mathbf{w}_j^s \oplus \mathbf{w}_j^t \tag{24}$$

$$= \sum_{i=1}^n \alpha_{ij}^s \mathbf{h}_i^s \oplus \sum_{i=1}^q \alpha_{ij}^t \mathbf{h}_i^t, \tag{25}$$

4 Experiments

In this section, we conduct experiments on the task of annotating source code. We first describe our dataset and data preparation steps, then we introduce our training configurations in detail, after that we present experimental results of our model and other baseline algorithms, finally, we show some persuasive generation examples to prove the practicality and readability of our model.

```
class StreamingBuffer(object):       # derive the class StreamingBuffer from the object base class.
  def __init__(self):                #   define the method __init__ with an argument self.
    self.vals = []                   #     self.vals is an empty list.
  def write(self, val):              #   define the method write with 2 arguments: self and val.
    self.vals.append(val)            #     append val to self.vals.
  def read(self):                    #   define the method read with an argument self.
    ret = b''.join(self.vals)        #     join elements of self.vals into a bytes string, substitute the result for ret.
    self.vals = []                   #     self.vals is an empty list.
    return ret                       #     return ret.
  def flush(self):                   #   define the method flush with an argument self.
    return                           #     return nothing.
  def close(self):                   #   define the method close with an argument self.
    return                           #     return nothing.
```

Fig. 5. An example of a code snippet with its annotation. The left part is a snippet of Python class, and the right part is its corresponding natural language annotation.

4.1 Dataset

For evaluating our model Code2Text effectively, we choose a high-quality Python-to-English dataset from [12].

Data Description. Python-to-English dataset contains the source code and annotations of Django Project (a Python web application framework). All lines of code are annotated with corresponding annotations by an engineer. The whole corpus contains 18,805 pairs of Python statements and corresponding English annotations, and we split it into a training set and a test set. The training set contains 16,000 statements, and we use it to train our Code2Text model. The rest 2,805 statements in the test set are used to evaluate the model performance. Figure 5 is an example code snippet from the training dataset.

Data Preparation. Since our model exploits hierarchical information of source code, we should first generate code ASTs by applying Python AST Parser[1] to each line of source code in the dataset. Fortunately, Python interpreter itself provides a built-in module called *ast* to help parse source code to its AST. Our model could work on other programming languages such as Java, C++, *etc.* as well if we apply their own open-source libraries for AST parsing[2,3].

For consistency, we need to apply the same preparatory operations to all the source code as follows:

1. For each line of source code, we parse it to an abstract syntax tree with a built-in *ast* module in the Python interpreter.
2. For each AST, we extract node tokens and put them into a tokens array.
3. Then, we index all the nodes and assign the parent's index for every token in the list. We assign index 0 to the root of the tree. The purpose of this step is to reconstruct the tree structure in our training and evaluation phase.

[1] https://docs.python.org/3/library/ast.html.
[2] https://github.com/javaparser/javaparser.
[3] https://github.com/foonathan/cppast.

Figure 4 shows an example of how we generate the tokens array and parent index list.

Finally, we extend our dataset by adding a new AST tokens array and an index list for each line of source code. We feed source code sequence into the sequential encoder and feed AST tokens sequence along with the parent index list to the tree encoder.

The goal of our preprocessing steps is decoding an AST into a nodes array(for node embedding) and a parent index array(for reconstruction). The order from the breadth-first search is not important here because the parent index of each node would help us reconstruct the AST tree. During the training process, we would like to accumulate structure information from leaves to root, so we use the parent index list to reconstruct an AST tree and compute root information with the help of parent index array by recursively applying tree-LSTM (As Fig. 2(b) shows).

4.2 Setup

In this part, we introduce our experiment setup, including compared methods and evaluation metric.

Our training objective is the cross-entropy, which maximizes the log probability assigned to the target words in the decoder process.

In the test phase, we have the same inputs in the encoder process for generating $\mathbf{h}_n^s$ and $\mathbf{h}_q^t$. In the decoder process, we predict with the START tag and compute the distribution over the first word y_1^p. We pick the argmax in the distribution and set its embedding vector as the next input y_1, and repeat this process until the END tag is generated. The whole generated sentence $\mathbf{y}^p$ will be our annotation result.

Table 1. Types of models, based on the kinds of features used.

Method	Rules	Seq. Info.	Tree Info.	Atte.
PBMT	√	–	–	–
Seq2seq	–	√	–	–
Seq2seq w/ attention	–	√	–	√
Code2Text w/o seq. info	–	–	√	√
Code2Text w/o attention	–	√	√	–
Code2Text	–	√	√	√

Compared Methods. To validate the improvement of our model, in this paper we compare Code2Text to following state-of-the-art algorithms (summarized in Table 1):

– PBMT [6,7,12]: PBMT is a statistical machine translation framework which uses the phrase-to-phrase relationships between source and target language pairs. Oda et al. apply PBMT to pseudocode generation task [12].
– Seq2seq [16]: Seq2seq model is commonly used in NMT tasks. It consists of encoder process and decoder process, while the encoder process encodes source code sequential information and decoder process learns a language model to predict annotations base on the summary of sequential information.
– Seq2seq w/ attention [9,19]: This version of seq2seq model incorporates attention mechanism which could improve the generation performance.
– Code2Text (w/o sequential encoder): This is one weak version of our method with only tree encoder and hierarchical attention mechanism.
– Code2Text (w/o attention): This weak version of our method combines sequential encoder and tree encoder with no attention mechanism.
– Code2Text: This is our method proposed in Sect. 3, which tries to improve the performance of automatic annotation generation.

Oda et al. [12] also proposed a model with AST information included, however, we have no comparability since we preprocess in different ways.

For all sequential encoders and decoders in both seq2seq models and our model, we use the one-layer LSTM network. The number of epoches is set to 30. All experiments were conducted under a Linux GPU server with a GTX 1080 device.

Metrics. In addition to the direct judgment from real cases, we choose BLEU (Bilingual Evaluation Understudy) score [13] to measure the quality of generated annotations for all the methods. BLEU is widely used in machine translation tasks for evaluating the generated translations. It calculates the similarity of generated translations and human-created reference translations. It is defined as the product of "n-gram precision" and a "brevity penalty" where n-gram precision measures the precision of length n word sequences and the brevity penalty is a penalty for short hypotheses. BLEU outputs a specific real value with range $[0, 1]$ and it becomes 1 when generated hypotheses completely equal to the references. We multiply the BLEU score by 100 in our experiments for display convenience.

4.3 Performance

Our Model vs. Other Models. From Table 2 we can conclude that our model Code2Text outperforms than other compared methods. For PBMT, we only compare BLEU-4 score due to the lack of the other three metrics in [12]. Code2Text has an obvious improvement than PBMT by around 1.6 times and outperforms better than attention seq2seq model since we incorporate hierarchical information in source code.

Table 2. Comparison *w.r.t* BLEU scores. Only BLEU-4 score reported for PBMT due to BLEU-1 to BLEU-3 are not available in source paper.

Models	BLEU-1	BLEU-2	BLEU-3	BLEU-4
PBMT	–	–	–	25.71
Seq2seq w/ atte.	54.11	46.89	42.02	38.11
Code2Text	**65.72**	**55.08**	**48.23**	**42.78**

Table 3. Comparison of BLEU Scores w/o Attention.

Models	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Seq2seq	34.29	28.46	24.61	21.56
+ attention	54.11	46.89	42.02	38.11
Code2Text	36.24	22.96	16.21	11.59
+ attention	**65.72**	**55.08**	**48.23**	**42.78**

Table 4. Effects of tree encoder under BLEU metric.

Models	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Seq2seq + att.	54.11	46.89	42.02	38.11
Code2Text w/o seq.	58.70	47.24	40.14	34.64
Code2Text	**65.72**	**55.08**	**48.23**	**42.78**

Effects of Attention Mechanism. Table 3 tells us how attention mechanism improves model performance. Whether in seq2seq or in our Code2Text, models with attention mechanism both perform better than who without attention mechanism. Meanwhile, the reason that Code2Text without attention mechanism has a lower BLEU score than seq2seq model is dual-encoder compresses more information than the sequential encoder, which makes it harder to capture important information if we do not have an alignment mechanism.

Effects of Tree Encoder. To evaluate the effects of tree encoder in our proposed model, Table 4 reveals that attention seq2seq model and our Code2Text model with only tree encoder has similar performance. Since this weak version of Code2Text neglects sequential encoder, it may not capture order information, which may result in worse performance under BLEU-3 and BLEU-4 than attention seq2seq model. However, our full version of Code2Text achieves the best score.

4.4 Case Study

We present four cases in Fig. 6. Each case has three corresponding annotations apart from the ground truth. For all the four cases, annotations generated by

Python code	status_code = 405
Ground truth	*status_code is an integer 405.*
Code2text	**status_code is an integer 405. ✓**
Code2text w/o attention	substitute name for self.name.
Seq2seq w/ attention	substitute 405 for status_code.

Python code	if exit_code < 0 :pass
Ground truth	*if exit_code is lesser than integer 0,*
Code2text	**if exit_code is smaller than integer 0, ✓**
Code2text w/o attention	equal to terminal_char , append the result to output
Seq2seq w/ attention	if exit_code is false,

Python code	return force_text (error)
Ground truth	*call the function force_text with an argument error, return the result.*
Code2text	**call the function force_text with an argument error, return the result. ✓**
Code2text w/o attention	call the function mark_safe with an argument data , return the result .
Seq2seq w/ attention	call the force_text with an argument error, return the result.

Python code	for i , line in enumerate (lines) :pass
Ground truth	*for every i and line in enumerated iterable lines ,*
Code2text	**for every i and line in enumerated iterable lines , ✓**
Code2text w/o attention	define the method __init__ with 3 arguments : self , unpacked list args and unpacked
Seq2seq w/ attention	**for every i and line in enumerated iterable lines , ✓**

Fig. 6. Cases of our Code2Text model. Our model could generate readable natural language annotations for various statements.

Code2text without attention mechanism have the least similarity to the ground truth, which meets the BLEU score evaluation results. The reason may be that LSTM performs worse due to the combination of source code tokens and AST tokens. The attention mechanism will help align the annotation words with the source tokens and AST tokens. For the first case and third case, although attention seq2seq model could generate reasonable annotations as well, our Code2Text captures the hidden keyword (*integer*, *function*) from source code AST and provides more accurate annotations. The fourth case reveals that Code2Text could generate complex expression, which is friendly to beginners.

We visualize our attention matrix α_{ij}^s and α_{ij}^t of the first and third cases in Fig. 7. For the left figure, our model captures the relationship between keyword *integer* and node tokens in AST (*int, n*). Since Python is a Weakly-Typed Language, the base type of a variable (such as integer, string, *etc.*) will be inferenced by the interpreter. Therefore, this type information could only exist in AST, that is why our model can generate keyword *integer*. In the same way, right figure exploits another two relationships. Keyword *function* corresponds with *func* and *argument* corresponds with (*expr*, args*).

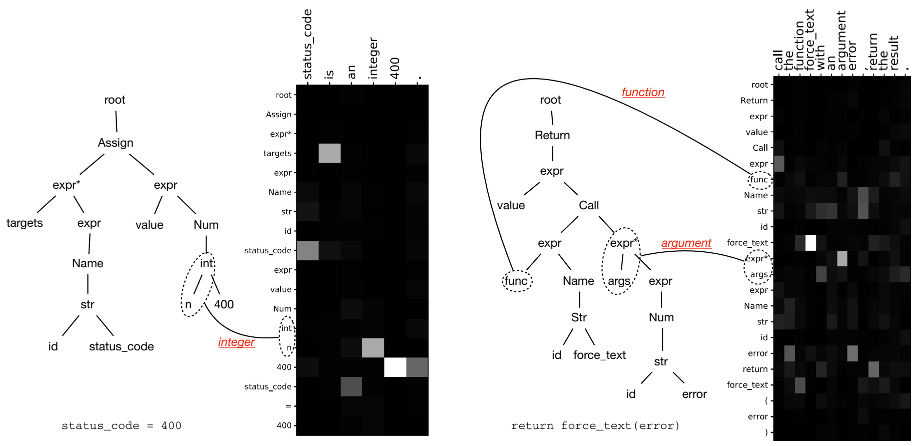


Fig. 7. Two sample alignments above refer to our first and third cases. The x-axis corresponds to the generated annotation, and the y-axis corresponds to the tokens from AST and source code. Each pixel shows the weight α_{ij}^{t} of the annotation of the j-th source word for the i-th target word (see Eq. 23), in grayscale (0: black, 1: white).

5 Related Work

In the early years, various rule-based models were explored by researchers. Sridhara et al. [14,15] focused on automatic comment generation for Java programming language. Their two works both concentrated on designing mapping rules between source code and code comment by hand and generating comments for Java methods by filling out pre-defined sentence templates. Moreno et al. [11] studied comment for Java Classes as well. Their model extracted the class and method stereotypes and used them, in conjunction with heuristics, to select key information to be included in the summaries. Then it generated code snippet summaries using existing lexicalization tools. However, there is a major limitation for the rule-based approach that it lacks portability and flexibility. When new rules that are never seen appear in the source code or we start a new project with another programming language, we have to manually update our rules table and sentence templates.

Later, most researchers tended to data-based approaches in recent years. Wong et al. [18] crawled code-description mappings from online Q&A websites at first, then output code comment by matching similar code segments. Therefore this model does not have a generalization. Haiduc et al. [3] created summaries for source code entities using text retrieval (TR) techniques adapted from automated text summarization [5].

Recently, some deep neural networks are introduced in the annotation generation task. In the task of pseudocode generation, Oda et al. [12] combines the rule-based approach and data-based approach. Their model updated the rules table automatically and generated pseudocode through n-gram language model.

However, its n-gram language model lacks explicit representation of long-range dependency which may affect generation performance. In 2017, Zheng et al. [19] applied attention sequence to sequence neural machine translation model on code summary generation which motivates our work. Allamanis et al. [1] treated source code as natural language texts as well, and learned a convolutional neural network to summarize the words in source code into briefer phrases or sentences. These models for code summary generation task do not consider the hidden hierarchical information inside the source code.

6 Conclusion

In this paper, we studied the problem of structure-aware annotation generation. We proposed a novel model, Code2Text, to translate source code to annotations by incorporating tree encoder and hierarchical attention mechanism. Experiment results showed that our model outperforms among state-of-the-art methods, and example cases prove the practicality and readability. Our model could be also extended to other programming languages easily with the specific parser.

Acknowledgements. This work is supported in part by the National Natural Science Foundation of China Projects No. U1636207, U1936213, the NSF under grants No. III-1526499, III-1763325, III-1909323, and CNS-1930941, the Shanghai Science and Technology Development Fund No. 19DZ1200802, 19511121204.

References

1. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: ICML 2016, New York City, NY, pp. 2091–2100 (2016)
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)
3. Haiduc, S., Aponte, J., Moreno, L., Marcus, A.: On the use of automated text summarization techniques for summarizing source code. In: WCRE 2010, Beverly, MA, pp. 35–44 (2010)
4. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
5. Jones, K.S.: Automatic summarising: the state of the art. Inf. Process. Manag. **43**(6), 1449–1481 (2007)
6. Karaivanov, S., Raychev, V., Vechev, M.: Phrase-based statistical translation of programming languages. In: SPLASH 2014, Portland, OR, pp. 173–184 (2014)
7. Koehn, P., Och, F.J., Marcu, D.: Statistical phrase-based translation. In: NAACL 2003, Edmonton, Canada, vol. 1, pp. 48–54 (2003)
8. Liu, X., Kong, X., Liu, L., Chiang, K.: TreeGAN: syntax-aware sequence generation with generative adversarial networks. In: ICDM 2018 (2018)
9. Luong, M.T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025 (2015)
10. Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: InterSpeech 2010, Makuhari, Japan (2010)

11. Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K.: Automatic generation of natural language summaries for Java classes. In: ICPC 2013, San Francisco, CA, pp. 23–32 (2013)
12. Oda, Y., et al.: Learning to generate pseudo-code from source code using statistical machine translation. In: ASE 2015, Lincoln, NE, pp. 574–584 (2015)
13. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: Bleu: a method for automatic evaluation of machine translation. In: ACL 2002, Philadelphia, PA, pp. 311–318 (2002)
14. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In: ASE 2010, Lawrence, KS, pp. 43–52 (2010)
15. Sridhara, G., Pollock, L., Vijay-Shanker, K.: Automatically detecting and describing high level actions within methods. In: ICSE 2011, Honolulu, HI, pp. 101–110 (2011)
16. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: NIPS 2014, Montreal, Canada, pp. 3104–3112 (2014)
17. Tai, K.S., Socher, R., Manning, C.D.: Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075 (2015)
18. Wong, E., Yang, J., Tan, L.: AutoComment: mining question and answer sites for automatic comment generation. In: ASE 2013, Palo Alto, CA, pp. 562–567 (2013)
19. Zheng, W., Zhou, H., Li, M., Wu, J.: Code attention: translating code to comments by exploiting domain features. arXiv preprint arXiv:1709.07642 (2017)