Developers often edit source code to add new features, fix bugs, or maintain existing functionality. Recent research has shown that these edits are often repetitive. Manually applying such repetitive edits can be tedious and error-prone. Thus, it is important to automate code changes, as much as possible, to reduce the developers' burden.

Recently there has been a lot of interest in using machine earning techniques to model and predict source code from
real world. However, modeling changes is different from modeling general code generation, since modeling changes is conditioned on the previous version of the code. In this work, we research whether machine learning models can capture the repetitiveness and statistical characteristics of code edits that developers apply daily. Such models can automate code changes, bug fixes, and other software-evolution-related tasks.

To this end, we use the general framework of Neural Machine Translation (NMT) to learn from past code changes
and suggest future edits.

In this work, our goal is to predict concrete code changes. To do that, we design an encoder-decoder-based NMT model that works in two steps. First, it learns and suggests structural changes in code using a tree-based translation model. Structural changes are suggested in the form of Abstract Syntax Tree (AST) modifications. Then, we concretize each code fragment generating the tokens conditioned on the AST generated in the first step. We combine these two models to realize CODIT, a code change suggestion engine, which takes a code fragment as input and generates potential edits of that snippet.

Synthesizing patches (or code in general) is a tough problem. When we view code generation as a sequence of
token generation problem, the space of the possible actions becomes too large. Existing statistical language modeling techniques (*e.g.,* Raychev *et al.* [11], Tu *et al.* [26]) endorse the action space with a probability distribution, which effectively reduces the action space significantly since it allows to consider only the subset of probable actions. The action space can be further reduced by relaxing the problem of concrete code generation to some form of abstract code generation, *e.g.,* generating code sketches, abstracting token names,
etc. For example, Tufano *et al.* reduce the effective size of the action space to 3:53 · 1010 by considering abstract token names. While considering all possible ASTs allowed by the language's grammar, the space size grows to 1:81·1035. In this work, a probabilistic grammar further reduces the effective action space to 3:18 · 1010, which is significantly lower than previous methods. Thus, the reduction of the action space allow us to search for code more efficiently.

Our experiments show CODIT achieves 15.81% patch suggestion accuracy in the top 10 suggestions. CODIT can also predict 33.61% of structural changes accurately within the top 10 suggestion. Further evaluation on CODIT's ability to suggest bug-fixing patches in Defects4J shows that CODIT suggests 16 complete fixes and 9 partial fixes out of 80 bugs in Defects4J.

NMT models are usually a cascade of an *encoder* and a *decoder*. In NMT, several improvements have been made over the base seq2seq model, such as *attention* [33] and *copying* [34]. Attention mechanisms allows decoders to automatically search information within the input sequence when predicting each target element. Copying, a special form of an attention mechanism, allows the model to directly copy elements of the input to the target. We employ both attention and copying mechanisms in this work.

While modeling the edit, CODIT first predicts the structural changes in the parse tree. For example, in Figure 1(a) CODIT first generates the changes corresponding to the subtrees with dark nodes and red edges. Next the structure is concretized by generating the token names (terminal nodes). This is realized by combining two models: (i) a tree-based model predicting the structural change followed by a (ii) a token generation model conditioned on the structure generated by the tree translation model.

Tree Translator: The tree translator is responsible for generating structural changes to the tree structure. A machine
learning model is used to learn a (probabilistic) mapping between *Tp* and *Tn*. First, a tree encoder, encodes *Tp* computing a distributed vector representation for each of the production rules *Tp* sequentially yielding the distributed representation for the whole tree. Then, the tree decoder uses the encoded representations of *Tp* to sequentially select rules from the language grammar to generate *Tn*.

Token Generator*:* The token generator generates concrete tokens for the terminal node types generated in the previous step. The token generator is a standard seq2seq model with attention and copying but constrained on the token types generated by the tree translator. To achieve this, the token generator first encodes the token string representation and the node type sequence from *Tp*. The token decoder at each step probabilistically selects a token from the vocabulary or copies one from the input tokens in *Tp*.