A Model Architecture Details

A.1 Implementation with ASDL

To implement the "dummy node" mechanism, we utilize the ASDL "field", which ensures the grammatical correctness of every edit. In ASDL, children of each tree node are grouped under different fields, and each field has a cardinality property (single, optional `?`, or sequential `*`) indicating the number of nodes they can accept as grammatically valid children. For single-cardinality fields that require exactly one child and optional-cardinality fields that require optionally zero or one child, we attach one dummy node when they do not have a child. For example, at $t = 1$ in Fig. 1(b), when the `MethodCall`-rooted subtree is deleted, we automatically attach a `Dummy` node to its parent field so the new node `ElementAccess` can be added by selecting `Dummy` and replacing it with the new node. Similarly, after deriving node `ElementAccess` at $t = 2$, we automatically add a `Dummy` node to each of its two fields (*i.e.*`obj` and `index`). For sequential-cardinality fields that accept multiple children, we always attach one dummy node as their rightmost child. Adding a new node in this case is implemented by selecting the right sibling of the target position and then inserting the new node to its left. For example, we extend the child list [`A`, `B`] to [`A`, `B`, `Dummy`]. Adding a new node to the left of `A` can then be achieved by selecting `A`, and adding to the right of `B` is done by selecting `Dummy`.

A.2 Tree Edit Decoder

Our edit decoder predicts an action a_t using three components: an operator predictor, a node selector, and a value predictor. At each time step t, the decoder's operator predictor first decides which operator $op_t \in \{\texttt{Delete}, \texttt{Add}, \texttt{CopySubTree}, \texttt{Stop}\}$ to apply. Next, for operators other than `Stop`, the node selector then predicts a node n_t from the tree on which to apply op_t. Finally, if $op_t \in \{\texttt{Add}, \texttt{CopySubTree}\}$, the value predictor further determines additional arguments of those operators (denoted as val_t). For `Add` actions, val_t denotes production rules or terminal tokens. For `CopySubTree`, val_t is the target subtree to copy. This is summarized as below:

$$p(a_t|\boldsymbol{s}_t) = p(op_t|\boldsymbol{s}_t)p(n_t|\boldsymbol{s}_t, op_t)p(val_t|\boldsymbol{s}_t, op_t, n_t).$$

Operator Prediction: The operator prediction is a 4-class classification problem. We calculate the probability of taking operator $op_t \in \{\texttt{Delete}, \texttt{Add}, \texttt{CopySubTree}, \texttt{Stop}\}$ at time step t as:

$$p(op_t|\boldsymbol{s}_t) = \text{softmax}(\boldsymbol{W}_{op}\boldsymbol{s}_t + \boldsymbol{b}_{op}),$$

where $\boldsymbol{W}_{op}$, $\boldsymbol{b}_{op}$ are model parameters.

Node Selection: Given a tree g_t, there could exist an arbitrary number of tree nodes. Therefore, we design the node selection module similar to a pointer network (Vinyals et al., 2015):

$$\begin{aligned}\boldsymbol{h}_{node,t} &= \tanh(\boldsymbol{W}_{node}[\boldsymbol{s}_t; \text{emb}(op_t)] + \boldsymbol{b}_{node}),\\ p(n_t|\boldsymbol{s}_t, op_t) &= \text{softmax}(\boldsymbol{h}_{node,t}^T \boldsymbol{n}_t),\end{aligned}$$

where $\text{emb}(op_t)$ embeds the previously selected operator op_t, $\boldsymbol{n}_t$ is the node representation, and $\boldsymbol{W}_{node}$, $\boldsymbol{b}_{node}$ are model parameters. The softmax is computed over all nodes $n_t \in g_t$.

Value Prediction: After deciding the target position (inferred from the selected node), adding a new node or subtree to the current tree can be viewed as expanding its parent node in typical tree-based generation tasks (Yin & Neubig, 2017; Rabinovich et al., 2017; Yin & Neubig, 2018). We thus adapt the tree-based semantic parsing model of Yin & Neubig (2018) as our value predictor.

Recall that the `Add` operator adds a new node to the tree by either applying a production rule ($val = rule$) or predicting a terminal token ($val = tok$), and the `CopySubTree` operator copies a subtree ($val = subtree$) to expand the tree. In all cases, we only consider candidates (*e.g.* production rules or subtrees) that satisfy the underlying grammar constraints. The prediction probability is also calculated via a pointer network in order to handle varying numbers of valid candidates in each decision situation:

$$\begin{aligned}\boldsymbol{h}_{val,t} &= \tanh(\boldsymbol{W}_{val}[\boldsymbol{s}_t; \boldsymbol{n}_t; \text{emb}(p_{n_t} \mapsto n_t)] + \boldsymbol{b}_{val}),\\ p(val_t|\boldsymbol{s}_t, op_t, n_t) &= \text{softmax}\big(\boldsymbol{h}_{val,t}^T \boldsymbol{W} \text{emb}(val_t)\big),\end{aligned}$$

Algorithm 1 DAGGERSAMPING

Require: $\langle f_\Delta, C_-, C_+ \rangle$ from training set, learning editor π_θ, expert policy π^*, $\beta \in [0, 1]$

Let $g_1 = C_-$.
Let $\pi' = \beta\pi^* + (1 - \beta)\pi_\theta$.
Sample a trajectory from $\pi'(f_\Delta, g_1)$.
Collect and return $\{\langle s, \pi^*(s)\rangle\}$ for all states s visited by π'.

Algorithm 2 POSTREFINESAMPING

Require: $\langle f_\Delta, C_-, C_+ \rangle$ from training set, learning editor π_θ, expert policy π^*

Let $g_1 = C_-$.
Sample a trajectory using $\pi_\theta(f_\Delta, g_1)$. Denote g_T as the output tree by the editor.
if $g_T \neq C_+$ **then**
    Sample a trajectory from $\pi^*(f_\Delta, g_T)$;
    Return $\{\langle s_t, \pi^*(s_t)\rangle | t \geq T\}$.
else
    Return empty collection.
end if

where $\boldsymbol{W}_{val}$, $\boldsymbol{b}_{val}$ and $\boldsymbol{W}$ are all model parameters, $\text{emb}(p_{n_t} \mapsto n_t)$ is the embedding of the edge type between the parent node p_{n_t} and the child n_t (*e.g.* `AssignStmt` $\xrightarrow{\text{right}}$ `ElementAccess`), and $\text{emb}(val_t)$ denotes the representation of the argument candidate: for production rules, it is their learned embedding; for terminal tokens, it is their word embedding; for subtree candidates, we use the representation of their root node.

B IMITATION LEARNING

We present DAGGERSAMPLING and POSTREFINESAMPLING in Algo. 1 and Algo. 2, respectively.

C DATASETS AND CONFIGURATIONS

For all datasets, we use the preprocessed version by Yin et al. (2019) for a fair comparison. The preprocessing includes tokenizing each code snippet and converting it into a AST.[6] For each $\langle C_-, C_+ \rangle$, we run a dynamic programming algorithm to search for the shortest edit sequence from C_- to C_+. The average length of gold edit sequences is 7.264 on GitHubEdits training set and 7.089 on C#Fixers.

Since surrounding contexts around the edited program are also provided in all datasets, we additionally allow the value predictor (§ 3.1) to copy a terminal token from either the input tree's code tokens or the contexts. To this end, we introduce another bidirectional LSTM encoder to encode the input code tokens as well as the contexts. The last hidden state is used to represent each token. The same design is also adopted in the two baseline editors.

For the encoder of our neural editor, the dimension of word embedding and the tree node representation is set to 128. The dimension of the bidirectional LSTM encoder for encoding input code tokens and contexts is set to 64. The hidden state for tracking tree history is set to 256 dimensions. In the decoder side, the dimensions of the operator embedding, the field embedding, the production rule embedding, and the hidden vector in value prediction are set to 32, 32, 128 and 256, respectively.

For a fair comparison, we follow Yin et al. (2019); Panthaplackel et al. (2020a) to encode a code edit into a real-valued vector of 512 dimensions. For our TreeDiff Edit Encoder, each edit action is encoded into a vector of 256 dimensions. The bidirectional LSTM also has a hidden state of 256 dimensions. When training Graph2Edit/Graph2Tree jointly with TreeDiff Edit Encoder, common parameters that are designed for both the neural editor and the edit encoder (*e.g.* the operator/field embedding) are shared.

In experiments, we reproduce and evaluate baselines by using implementations kindly provided by their authors. This includes testing the baseline editors under exactly the same setting as they were tested in their original paper (*e.g.* decoding using beam search of size 5 for Graph2Tree and 20 for CopySpan).

[6] The ASDL grammar we used for C# can be found at: `https://raw.githubusercontent.com/dotnet/roslyn/master/src/Compilers/CSharp/Portable/Syntax/Syntax.xml`.

Table 4: The nearest neighbors of given edit pairs based on their edit representations.

Example 1	Example 2
C_-: `BoundsCheck(VAR0, VAR1);` C_+: `BoundsCheck(VAR1, VAR0);`	C_-: `var VAR0=GetEtagFromRequest();` C_+: `var VAR0=GetLongFromHeaders(LITERAL);`
Graph2Tree – Seq Edit Encoder	
▶ C_-: `ReleasePooledConnectorInternal(VAR0, VAR1);` C_+: `ReleasePooledConnectorInternal(VAR2, VAR0);`	▶ C_-: `var VAR0=new ProfileConfiguration();` C_+: `var VAR0=new Profile(LITERAL);`
▶ C_-: `UngetPooledConnector(VAR0, VAR1);` C_+: `UngetPooledConnector(VAR2, VAR0);`	▶ C_-: `var VAR0=PrepareForSaveChanges();` C_+: `var VAR0=PrepareForSaveChanges(null);`
▶ C_-: `VAR0.Warn(LITERAL, VAR1);` C_+: `VAR0.Warn(VAR1, LITERAL);`	▶ C_-: `bool VAR0=true;` C_+: `bool VAR0=CanBeNull(VAR1);`
Graph2Tree – TreeDiff Edit Encoder	
▶ C_-: `InternalLogger.Error(LITERAL, VAR0);` C_+: `InternalLogger.Error(VAR0, LITERAL);`	▶ C_-: `var VAR0=new ProfileConfiguration();` C_+: `var VAR0=new Profile(LITERAL);`
▶ C_-: `VAR0.Warn(LITERAL, VAR1);` C_+: `VAR0.Warn(VAR1, LITERAL);`	▶ C_-: `CalcGridAreas();` C_+: `SetDataSource(VAR0, VAR1);`
▶ C_-: `AssertEqual(VAR0.Value, LITERAL);` C_+: `AssertEqual(LITERAL, VAR0.Value);`	▶ C_-: `VAR0=new Win32PageFileBackedMemoryMappedPager();` C_+: `VAR0=new Win32PageFileBackedMemoryMappedPager(` `LITERAL);`
Graph2Edit – Seq Edit Encoder	
▶ C_-: `ReleasePooledConnectorInternal(VAR0, VAR1);` C_+: `ReleasePooledConnectorInternal(VAR1, VAR0);`	▶ C_-: `var VAR0=new ProfileConfiguration();` C_+: `var VAR0=new Profile(LITERAL);`
▶ C_-: `UngetPooledConnector(VAR0, VAR1);` C_+: `UngetPooledConnector(VAR2, VAR0);`	▶ C_-: `VAR0.Dispose();` C_+: `VAR0.Close(VAR1);`
▶ C_-: `ReportUnusedImports(VAR0, VAR1, VAR2);` C_+: `ReportUnusedImports(VAR2, VAR0, VAR1);`	▶ C_-: `VAR0=VAR1(VAR2);` C_+: `VAR0=GetSpans(VAR2, VAR1);`
Graph2Edit – TreeDiff Edit Encoder	
▶ C_-: `VAR0.Warn(LITERAL, VAR1);` C_+: `VAR0.Warn(VAR1, LITERAL);`	▶ C_-: `var VAR0=new ProfileConfiguration();` C_+: `var VAR0=new Profile(LITERAL);`
▶ C_-: `InternalLogger.Error(LITERAL, VAR0);` C_+: `InternalLogger.Error(VAR0, LITERAL);`	▶ C_-: `VAR0=Thread.GetDomain().DefineDynamicAssembly(VAR1,` ↪ `AssemblyBuilderAccess.Run);` C_+: `VAR0=Thread.GetDomain().DefineDynamicAssembly(VAR1,` ↪ `AssemblyBuilderAccess.RunAndSave, LITERAL);`
▶ C_-: `AssertEqual(VAR0.Value, LITERAL);` C_+: `AssertEqual(LITERAL, VAR0.Value);`	▶ C_-: `new DocumentsCrud().EtagsArePersistedWithDeletes();` C_+: `new DocumentsCrud().PutAndGetDocumentById(LITERAL);`

For the supervised learning, we train our Graph2Edit for 30 epoches on GitHubEdits training set, where the best model parameters are selected based on the editor's cross entropy loss on dev set.

D Additional Experimental Results

Tab. 4 shows the nearest neighbors of given edit pairs from GHE dev set, based on the cosine similarity of their edit representations $f_\Delta(C_-, C_+)$ calculated by different edit encoders. The edit in Example 1 means to swap function arguments (*e.g.* from "`(VAR0,VAR1)`" to "`(VAR1, VAR0)`"). Intuitively such structural changes can be easily captured by our tree-level edit encoder. This is consistent with our results, which show that, for both Graph2Tree and Graph2Edit, TreeDiff Edit Encoder learns more consistent edit representations for this edit, while Seq Edit Encoder may confuse it with edits that replace the original argument with a new one (*e.g.* modifying "`(VAR0,VAR1)`" to "`(VAR2,VAR0)`"). Our proposed edit encoder can also generalize from literals (*e.g.* swapping between "`(VAR0,VAR1)`") to more complex expressions (*e.g.* swapping between "`(VAR0.Value, LITERAL)`"). On the other hand, when the intended edits can be easily expressed as token-level editing (*e.g.* inserting an argument token), the two edit encoders perform comparably, as shown in Example 2. However, we still observe that TreeDiff Edit Encoder works better at interpreting the editing semantics of code snippets with complex structures (*e.g.* more complex edit pairs are retrieved).