GRAPH REPRESENTATIONS IN MULTI-MODAL INPUT FOR CODE PATCHING

Micah Cheng, & Jordan Dreibelbis
Department of Computer Science
Xinhao Su
Department of Computer Engineering
Columbia University
New York, NY 10027, USA
`{xs2413, mwc2147, jed2205}@columbia.edu`

ABSTRACT

Graph representations of code have recently shown promise when applied to AI4SE tasks. Compared to simple sequences, graphs are able to capture more semantics of code using control or data flow edges, and allow models to utilize the rich structure of code to better learn. We present a proof of concept multi-modal model that utilizes three different modalities to generate edits for the given code. We use edit location, natural language commit messages, and overall code context represented as a graph. We show that there is no improvement over a sequential textual based model. Thus there is no indication whether a graph based model will improve the performance of multi modality input towards code editing. However, we do believe our performance can greatly improve by more sophisticate tuning on the model configuration.

1 INTRODUCTION

Code representation is a difficult task, as code is rich in structural and semantic information when compared to natural language. At first, code representations borrowed from their natural language counter parts for representation. However there have been many attempts in recent years to find better ways of representation, ranging from shallow, textual structures such as a simple sequence of tokens, or parse trees Bielik et al. (2016), to more complex structural graphs such as abstract syntax trees (AST) Allamanis (2018), augmented abstract syntax trees Allamanis et al. (2018), and code property graphs Yamaguchi et al. (2016). These code-specific graph representations have emerged as a promising way to improve code representation on downstream tasks.

Machine learning has shown promise in source code editing. One interesting field in this area is the improvement of input representation for the model, utilizing the rich structure of the code and capturing the vital semantic information such that the model can propose relevant, bug-free, and complete edits. Our work builds upon recent work in modeling code for source code editing when paired with multiple modalities. MODIT, proposed by Chakraborty & Ray (2021), introduced the use of adding additional representations of information: guidance, code, and context, as input features to improve performance of learning to edit source code. Specifically, MODIT requires the user to provide a natural language description for the edit's intention as guidance, a code snippet that is waiting to be edited, and code context that is provided by the user which include the surrounding method body or all the relevant lines of code. In MODIT's implementation, both intent-to-edit code, guideline, and the context are tokenized by a sentence-piece tokenizer that divides every token into sequence of sub-tokens. MODIT proved this guideline is essential to curate the user's intentions and compared its performance by using commit messages as guideline and no guideline at all. In the real world, commit messages tend to be noisy and may not reflect the true intention behind the commit, but their empirical study showed using commit messages as a guideline will outperform no guideline at all. However, we noted their representation of the model uses simple textural sequences and misses the opportunity to exploit the structural information of the code. Therefore, we place our attention on improving the representation of intent-to-edit code and the context.

2 Related Work

GGNN, proposed by Allamanis et al. (2018), aimed to perform two tasks: VarNaming and VarMisused. To achieve high accuracy of these tasks, the model eventually needs to distinguish the purpose of the variable and then inferring the usage of the variables(i.e. How and where a given variable is used). Nevertheless, the first task may be able to be learned from a sequence of tokens, but the second task requires the model to learn how a variable is declared, assigned, or used in order to understand the role and possible value of the variable (i.e. Data flow of the program). Hence, GGNN contributed a new way to represent a program, a AST with control flow edges and data flow edges. This concept of program graph is fundamental to our work. In GGNN implementation, AST consists of two types of nodes: syntax nodes and syntax tokens, where syntax nodes represent the "grammar" part of the AST (any nodes that are not leaves) and the syntax tokens represent the actual code tokens. Take the following line of code as example: "if (a) c = b; " The syntax node will be "IfStatement" or "AssignmentExpression", and the syntax token will be "If" "(", "a", ")", "b", "=" and "c". In addition, the control flow edges and data flow edges will only be connected between the syntax tokens.

GREAT, proposed by Hellendoorn et al. (2020), extends the work of GGNN by showing the model can benefit from combining both local and global information. This is done using a "sandwich" model, adding a RNN or transformer layer to the GGNN, meaning the global sequential information is also available along with the structural information. In terms of the representation of the code, GREAT also used a program graph that was proposed by GGNN, but it removed the syntax node and kept only the syntax token. It argues that the structural relationship of the program is naturally represented by control flow, allowing for far fewer nodes, and saving space and time to train the model. Our work also adapts GREAT's version of the program graph.

PLUR, proposed by Chen et al. (2021), introduced a unifying framework to incorporate 16 well-known code training tasks from different previous works into the model such that the rich output can be used for many downstream tasks. To adapt a variety of tasks, PLUR standardized a graph based input and then "translating" into different representations to accommodate different tasks. Our work seeks the opportunities to utilize such techniques of "translation" to encode more information into our representation of the context.

3 Methods

3.1 MODIT baseline

Recent work from Chakraborty & Ray (2021) has shown that using a modal input representation improves performance in source code editing with their MODIT model. This multi-layer encoder-decoder model represents input using three modalities: Code that needs to be edited (e_p), natural language guidance (G), and context (C). For our purposes, we will be using this architecture as our baseline model. Since we are focused on investigating possible input representation improvements, we leave the pre-processing, transformers, and output-generation the same as is presented in MODIT. Instead, we look at representing G code to be edited, and C context using a graph-based encoding.

3.2 Graph-Based Encoding from the Program Graph, GREAT, PLUR

Program Graph from Allamanis et al. (2018), has proved the graph-based encoding is capable of utilizing the rich structure of the code and capturing the vital semantic information. Our approach will adapt this idea. In additional to that, GREAT(Hellendoorn et al. (2020)) model argues since the structural relationship of the program is naturally presented by control flow, the syntax nodes are redundant in a sense that they don't contribute any additional inductive bias than syntax token, such that only keeping syntax tokens is sufficient. Our method should be able to generate these two kinds of encoding, and then conducting a ablation study on this claim.

However, it is important to know that we will not try to emulate exactly as same as the approaches proposed by Allamanis et al. (2018) and Hellendoorn et al. (2020). On one hand, GREAT does not open-source their detailed implementation due to the license issue. On the other hand, both approaches of program graph and GREAT are subject to different languages, written in different languages (E.g. program graph was coded in C#. The main reason of that is possibly the fact that their dataset is a corpus of C#, and they need the tools from C# community to handle miscellaneous implementations' details on C# source code, like AST parser for C#.). Therefore we will approximate the structure of the original data representation and encoding, inferring the favorable inductive bias from original encoding.

Our graph-based encoding, denoted as G, consisting of Nodes V and Edges E, such that $G = (V, E)$. We will use third-party library to help us parse the Java source code. For each AST node, we construct a node u with *type* to distinguish whether it is a syntax node or syntax token, *token_value* to store the actual value of a syntax token, and a unique integer *id* that we assign based on the linear order of a given AST node. We denote that:

$$u = (type,\ token\_value,\ id), u \in V$$

Further, we need to build edges to represent the AST edge, control flow and data flow. Each edge, e, will consists of the source node *id*, target node *id*, and a *edge_type* to indicate whether is is the AST, control flow, or data flow edges. We denote as:

$$e = (source\_id,\ target\_id,\ edge\_type), e \in E$$

To prove the effectiveness of the graph-based encoding, we plan to implement a set of ablation studies. A variety of datasets will be trained and evaluated based on the MODIT's original evaluation. These datasets include *AST-Only* which contains both AST token and AST leaves along with the AST edges; *Leaves-Only* that contains only the AST leaves and control flow and data flow edges (which is adapted from Hellendoorn et al. (2020)); *CFG-Only* that only include the the AST leaves and control flow edge; *DFG-Only* that only include the the AST leaves and data flow edges;.

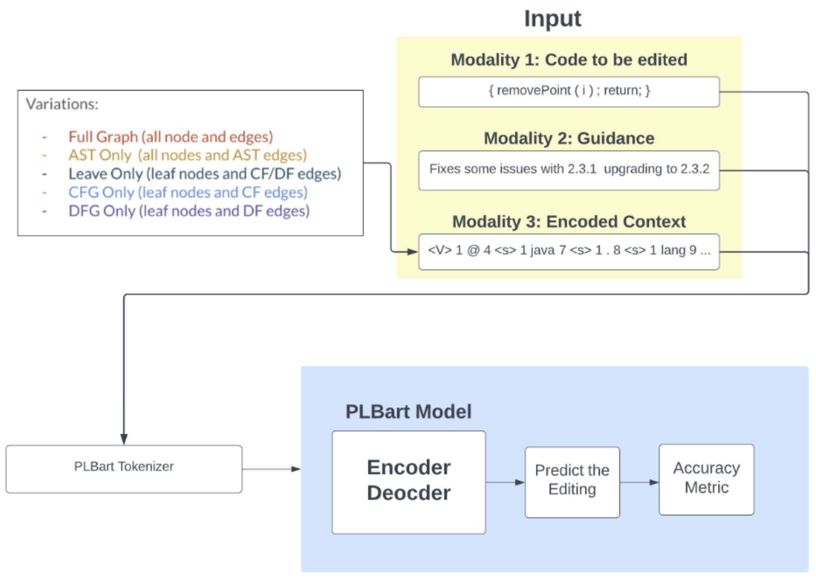

Input
Modality 1: Code to be edited
{ removePoint (i) ; return; }
Modality 2: Guidance
Fixes some issues with 2.3.1 upgrading to 2.3.2
Modality 3: Encoded Context
<V> 1 @ 4 <s> 1 java 7 <s> 1 . 8 <s> 1 lang 9 ...
Variations:
- Full Graph (all node and edges)
- AST Only (all nodes and AST edges)
- Leave Only (leaf nodes and CF/DF edges)
- CFG Only (leaf nodes and CF edges)
- DFG Only (leaf nodes and DF edges)
PLBart Tokenizer
PLBart Model
Encoder Deocder
Predict the Editing
Accuracy Metric

Figure 1: Pipeline of the multi-modal model used for experiments

3.3 Transformer Model

Following MODIT, we pre-process our three modal inputs by generating consolidated multi-modal input (X) as sequences separated by a special $< s >$ token. $X = e_p < s > G < s > C$. The first input modality is the Patch that is being editing, denoted as e_p; A Guidance that guideline the purpose of the patch, denoted as G, which Modit has proven the guidance is essential and effective for model understand the underlying intention of the code.Chakraborty & Ray (2021). In our case, the guidance will be the commit message. The third modality is the Context with respect to the

Patch, denoted as C. Our representations of C are graph-based in this instance. The input X will be tokenized using the PLBartTokenizer proposed by PLBART Ahmad et al. (2021) that divides each token into a sequence of sub-tokens.

This input sequence is fed into a six layer encoder, computing self-attention of each token at each layer, l as:

$$R_l^e(x_j) = \sum_{j=i}^{n} a_{i,j} * R_{l-1}^e(x_j)$$

The encoder output is then fed into a six layer decoder containing two modules of self-attention and cross-attention, where the self-attention follows a similar equation as the encoder. The cross-attention is computed as:

$$D_l(y_i) = \sum_{j=i}^{n} \alpha_{i,j}^l * R_6^e(x_j)$$

Here, $\alpha_{i,j}^l$ is the attention weight between output sub-token and input sub-token. The output, $D_l(y_i)$ is projected to the output vocabulary. Both the encoder and decoder are initialized using the pre-trained weights from PLBART.

Output is then generated by the decoder until it reaches an end of sequence token $< /s >$, predicted by beam search. This output sequence is then decoded back into Java code after stripping the end of sequence token. For a more detailed explanation of MODIT architecture, we refer readers to Chakraborty & Ray (2021).

4 Evaluation

4.1 Data Preparation

We use the $B2F_m$ dataset proposed by Tufano et al. (2018) which consist of Java changes from Github, alongside the matching commit message. We extract the input modalities and expected output to train the model. For each code context, we use Tree-sitter to parse the code into an AST, adding edges for Control Flow, AST Flow, Last-Read, and Last-Write depending on the variation of the graph type. We do a pre-order DFS traversal and label each node with an id corresponding to its place, to keep the sequential information of the graph. We then combine a serialized sequence of the tree with the guidance of the commit message and the code change to create the three input modalities.

4.2 Training

We use a pre-trained PLBartTokenizer and PLBartForConditionalGeneration model Ahmad et al. (2021) finetuned for our purposes. The model uses an Adam optimizer with learning rate of $5e^{-}5$, and has an an input size of 1024. We train on 4 epochs and test on a validation set after each performance. We use language modeling loss along with the PLBart default configurations in our training.

4.3 Evaluation

We evaluate using the top-1 accuracy, meaning the output code must exactly match the expected code. The consequences of this strict evaluation metric are discussed later in the paper. We use the original MODIT model as baseline, and show comparisons on fine-tuned versions of CodeBERT, GraphCodeBERT, and CodeGPT which use the original MODIT dataset of sequential tokens, displayed in Table 1.

5 Results

5.1 Quantitative Evaluation

Table 1 shows the evaluation results for each model. We can see our fine-tuned graph-based models perform worse than the fine-tuned sequential models. The *CFG-Only*, *DFG-Only*, and *Leaves-Only*

Table 1: Top-1 accuracy for each model. Sequential models all use multi modal input with context represented by sequential tokens. Graph models represent this as a graph with different types of edges.

Input Type	Model Name	Accuracy (%)
Sequential	CodeBERT	17.13
	GraphCodeBERT	18.31
	CodeGPT	17.64
	MODIT	**23.02**
Graph-based	CFG Only	00.80
	DFG Only	00.93
	AST Only	03.37
	Leaves Only	00.73

graphs perform about the same, just below 1%. However the simple AST reaches the best accuracy of 3%.

5.2 Qualitative Evaluation

We performed a case study on several of the predictions made by our *Leaves-Only* model and detail each issue in Appendix A. We see in Figure 3 that the model misunderstood the <SEP> token, learning it into the vocabulary of potential words. It also confused some graph properties and words, as shown in Figure 4, where the numbers were likely learned from the graph encoding.

The model also has a weak understanding of Java syntax, which might be a result of training on a low number of epochs. In Figure 5 we see multiple instances of incorrect syntax. Figure 6 illustrates other behavior, for instance the model being unable to predict variable names if the name is unseen or rarely seen in its vocabulary. We have included several commit changes that do not contain a bug fix and paired them with a NO BUG output. However, for every one of these examples, the model outputs $\{EMPTY >$ instead of the correct $< EMPTY >$.

6 Attempted Improvement

We are planing to make some improvements on the encoding. First, since we didn't observe the significant favorable inductive bias from the the Control Flow edge, we can remove it from the encoding. We suspected that the node id of the original encoding is already highlight the control order of the code tokens, such that the control flow edges are only providing very little information to the model. On the other hand, since we obverse the significant better performance in AST-Only, we argue that the AST edge did help the model to understand the syntax of the code, improving the model's understanding of the syntax. We also including the Last-Read and Last-Write since it does show promotion on semantics relevance of the model.

Second, to mitigate the divergence between the representation of the Patch and Context, where the Patch will be encoded as plain sequence of token and the Context will be encoded as program graph, we can bind each Patch token to the corresponding node inside the Context by assigning node id to each Patch Token during the prepossessing. For example:

```
// Before
{ return Null; }

// After
{ 72 return 75 Null 78 ; 79 } 81
```

Figure 2: Lightweight encoding on Patch

We believe this modification will also highlight the Patch inside the graph-based Context, reinforcing the model ability to distinguish the differences between Code Token and Graph Properties.

Third, we assign one more extra signal to each Node to further augment the indication of the Patch inside the graph-based context. For each node that inside in the range of the Patch, we assign 1 to highlight it is a part of the Patch, 0 otherwise:

$$u = (type,\ token\_value,\ id), u \in V_{old}$$
$$u^{`} = (type,\ token\_value,\ id,\ is\_patch), u^{`} \in V_{new}$$

7 Conclusion

In this paper, we present a proof of concept approach to using graphs on multi-modal input for code editing. Our evaluation shows some insight into the use of graphs on sequential models, as well as the impact that data flow, control flow, and syntax edges have on code editing using this multi modal approach. It is unclear if the poor performance of the models is from lack of training or the result of the graph representation. In the future, we want to explore more extensive training, as well as an analysis of code similarity of the results, instead of top-1 exact accuracy. All code can be found at https://github.com/mwcheng21/knowledge-representation.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *NAACL*, 2021.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *ICLR*, 2018.

Premkumar Devanbu Charles Sutton Allamanis, Earl T Barr. A survey of machine learning for big code and naturalness. *CSUR*, 2018.

Pavol Bielik, Veselin Raychev, and Vechev Martin. Phog: probabilistic model for code. *ICML*, 2016.

Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. *ICLR*, 2021.

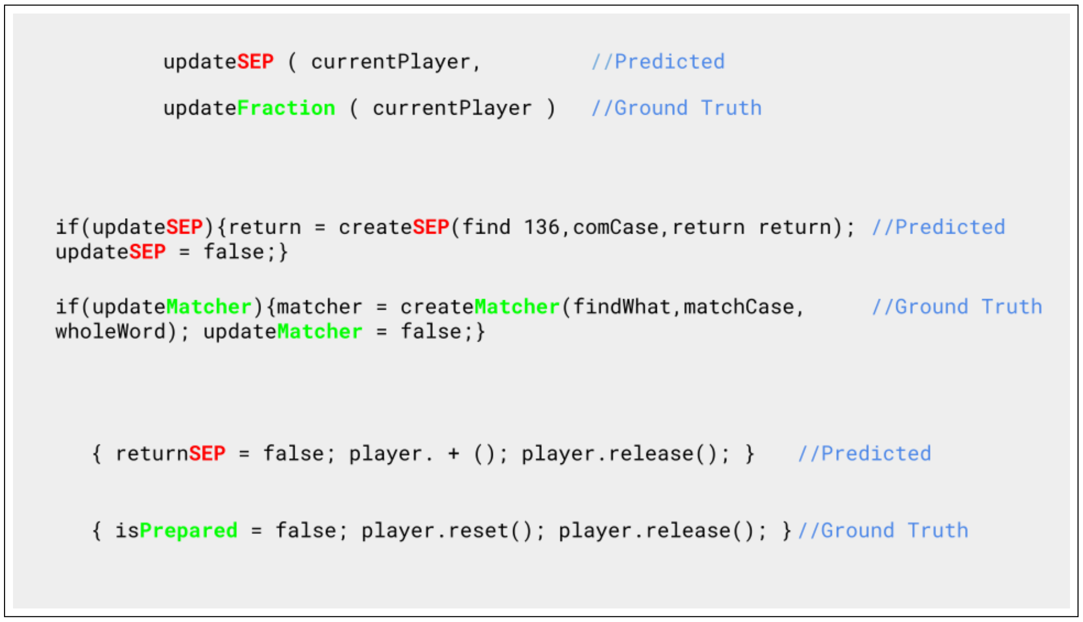
Zimin Chen, Vincent Josua Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Managol, Daniel Tarlow, and Moitra Subhodeep. Plur: A unifying, graph-based view of program learning, understanding, and repair. *ICLR*, 2021.

Vincent J. Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. Global relational models of source code. *ICLR*, 2020.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *CoRR*, abs/1812.08693, 2018.

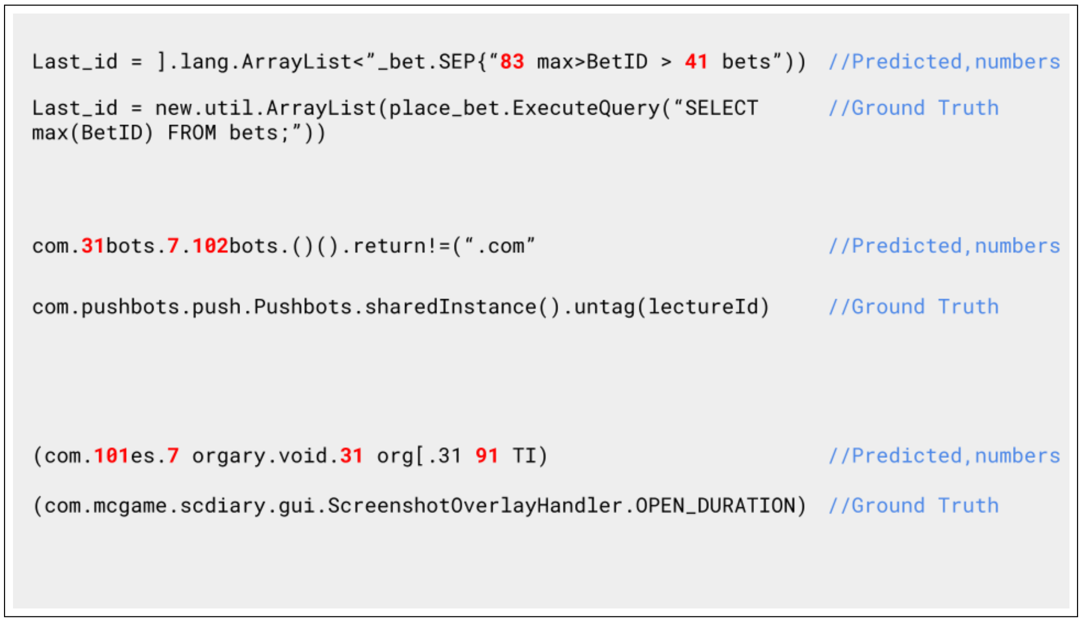
Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. *IEEE*, 2016.

A APPENDIX A



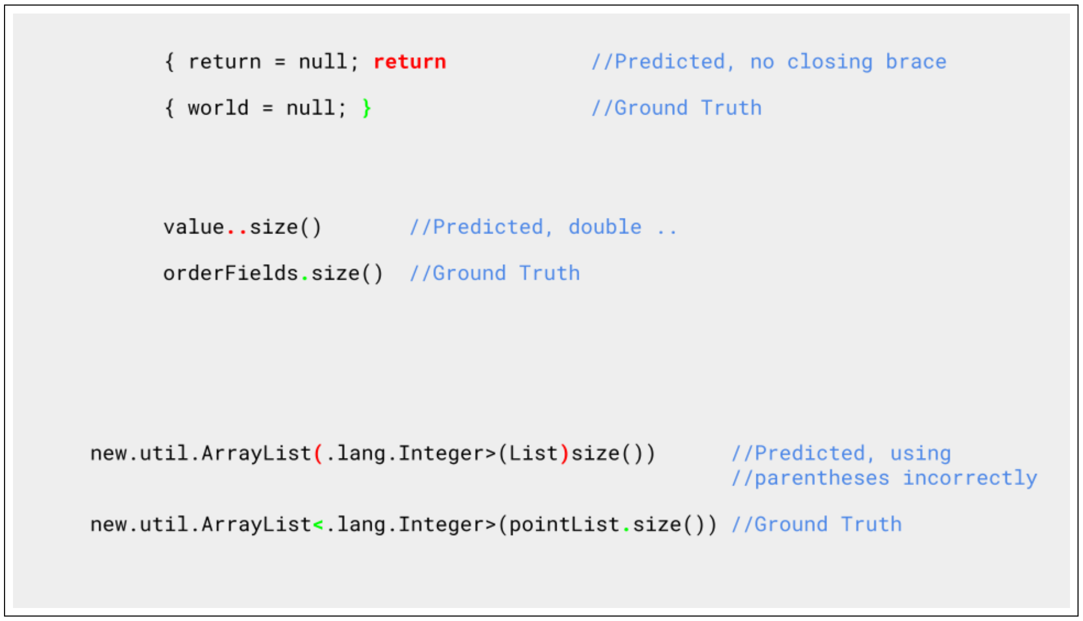
```
updateSEP ( currentPlayer,          //Predicted

updateFraction ( currentPlayer )    //Ground Truth


if(updateSEP){return = createSEP(find 136,comCase,return return); //Predicted
updateSEP = false;}

if(updateMatcher){matcher = createMatcher(findWhat,matchCase,     //Ground Truth
wholeWord); updateMatcher = false;}


{ returnSEP = false; player. + (); player.release(); }    //Predicted

{ isPrepared = false; player.reset(); player.release(); }//Ground Truth
```

Figure 3: Predictions that misunderstand the <SEP> token, with the prediction on top and ground truth underneath. In each example, the model predicts the camel cased word to use the word SEP.



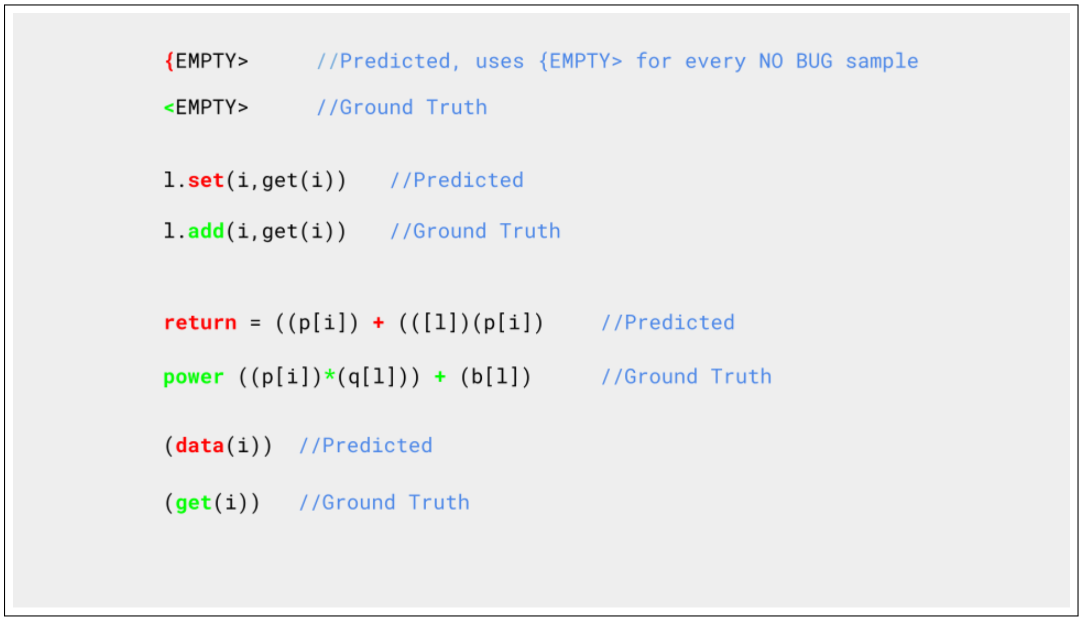
```
Last_id = ].lang.ArrayList<"_bet.SEP{"83 max>BetID > 41 bets")) //Predicted,numbers

Last_id = new.util.ArrayList(place_bet.ExecuteQuery("SELECT     //Ground Truth
max(BetID) FROM bets;"))


com.31bots.7.102bots.()().return!=(".com"                       //Predicted,numbers

com.pushbots.push.Pushbots.sharedInstance().untag(lectureId)    //Ground Truth


(com.101es.7 orgary.void.31 org[.31 91 TI)                      //Predicted,numbers

(com.mcgame.scdiary.gui.ScreenshotOverlayHandler.OPEN_DURATION) //Ground Truth
```

Figure 4: Predictions that confuse the words and graph properties, with the prediction on top and ground truth underneath. Each prediction includes some numbers, likely learned from the graph encoding which labels the position of each node, as well as the edge type with an integer.



```
{ return = null; return          //Predicted, no closing brace

{ world = null; }                //Ground Truth


value..size()       //Predicted, double ..

orderFields.size()  //Ground Truth


new.util.ArrayList(.lang.Integer>(List)size())      //Predicted, using
                                                    //parentheses incorrectly

new.util.ArrayList<.lang.Integer>(pointList.size()) //Ground Truth
```

Figure 5: Predictions with weak Java syntax, with the prediction on top and ground truth underneath. Example 1 has no closing ”}”, example 2 has a double ”.”, and example 3 predicts a ”)” instead of ”.”.



```
{EMPTY>     //Predicted, uses {EMPTY> for every NO BUG sample

<EMPTY>     //Ground Truth

l.set(i,get(i))    //Predicted

l.add(i,get(i))    //Ground Truth

return = ((p[i]) + (([l])(p[i])     //Predicted

power ((p[i])*(q[l])) + (b[l])      //Ground Truth

(data(i))  //Predicted

(get(i))   //Ground Truth
```

Figure 6: Predictions with weird behavior, with the prediction on top and ground truth underneath. The model correctly predicts all NO BUG samples, but with the incorrect output. For unseen vocab, the model defaults to using ”return” instead of another applicable word.