Deep Learning On Code with an Unbounded Vocabulary

Milan Cvitkovic*
Caltech
Pasadena, CA, USA
Amazon AI
Seattle, WA, USA
mcvitkov@caltech.edu

Badal Singh
Amazon Web Services
Seattle, WA, USA
sbadal@amazon.com

Anima Anandkumar
Caltech
Pasadena, CA, USA
Amazon AI
Seattle, WA, USA
anima@caltech.edu

A major challenge when using techniques from Natural Language Processing for supervised learning on computer program source code is that many words in code are neologisms. Reasoning over such an unbounded vocabulary is not something NLP methods are typically suited for. We introduce a deep model that contends with an unbounded vocabulary (at training or test time) by embedding new words as nodes in a graph as they are encountered and processing the graph with a Graph Neural Network.

1 Introduction

Computer program source code is an abundant, accessible, and important form of data. But despite the similarities between natural language and source code, deep learning methods for Natural Language Processing (NLP) have not been straightforward to apply to learning problems on source code like code completion and automated variable name generation.[1]

There are many reasons for this, but two central ones are:

1. *Code is extremely sensitive to syntax.* Natural language sentences can be messy and still get their point across. It is precisely this lack of rigid structure that makes learning necessary for understanding natural language in the first place. But this lack of structure makes source code a challenge for NLP methods: a tiny change in code syntax can result in a huge change in semantics. E.g. one tab too few in Python can completely change the contents of a `for` loop.
2. *Code is written using an unbounded vocabulary.* Natural language is mostly composed of words from a large but fixed vocabulary. Standard NLP methods can thus perform well by fixing a large vocabulary of words before training, labeling the few words they encounter outside this vocabulary as "Unknown". But in code every new variable or method declared requires a new, often abstruse, name. A model must be able to reason about these neologisms to understand code.

Prior work on deep learning for source code has attempted to address the syntax–sensitivity issue. A common strategy in these works is to represent source code as an Abstract Syntax Tree (AST) rather than as linear text.

In this work we extend such AST–based representation strategies to attempt to address the unbounded–vocabulary issue. The model we present does this by representing vocabulary words as nodes in the AST graph that are added as they are encountered, essentially using the graph as an unbounded, relational vocabulary cache. This, in principle, gives the model a way to reason during testing over words and phrases it never encountered during training.

*Corresponding author

Submitted to:
CAV 2018

2 Related Work

Representing Code as a Graph

Given their prominence in the study of programming languages, ASTs and parse trees are a natural choice for representing code and have been used extensively. Often models consume ASTs by linearizing them (usually with a depth–first traversal) [3, 20, 18], but they can also be processed by deep learning models that take graphs as input, as in [31, 8] who use Recursive Neural Networks (RveNNs) [11] on ASTs. RveNNs are models that operate on tree–topology graphs, and have been used extensively for language modeling [29] and on domains similar to source code, like mathematical expressions [33, 4]. They can be considered a specialized type of Message Passing Neural Network (MPNN) [10]: in this analogy RveNNs are to Belief Propagation as MPNNs are to Loopy Belief Propagation. ASTs also serve as a natural basis for models that generate code as output, as in [23, 32, 26, 8].

Data–flow graphs are another type of graphical representation of source code with a long history [17], and they have occasionally been used to featurize source code for machine learning [7].

Most closely related to our work is the recent work of [2], on which our model is heavily based. They combine the data–flow graph and AST representation approaches by representing source code as an AST augmented with extra labeled edges indicating semantic information like data– and control–flow between variables. These augmentations yield a directed graph rather than just a tree, so they (and we) use a variety of MPNN called a Gated Graph Neural Network (GGNN) [19] to consume the augmented AST.

Graph-based models that are not based on ASTs are also sometimes used for analyzing source code, like Conditional Random Fields for joint variable name prediction in [27]

Reasoning about Neologism

The question of how to gracefully handle out–of–vocabulary words is an old one in NLP. Character–level embeddings are a typical way deep learning models handle this issue, whether used on their own [14], or in conjunction with word–level embedding Recursive Neural Networks (RNNs) [22], or in conjunction with an *n*–gram model [6]. Another approach is to learn new word embeddings on–the–fly from context [16].

In terms of producing outputs over variable–sized input and outputs, attention-based pointer mechanisms were introduced in [30] and used to great effect in NLP in [24], whose sentinel pointing model we take inspiration from in the Variable Naming task below (although they use a fixed vocabulary).

Using graphs to represent arbitrary collections of entities and their relationships for processing by deep networks has been widely used [13, 5, 25, 21], but to our knowledge we are the first to use a graph–building strategy for reasoning (at train *and* test time) about an unbounded vocabulary of words.

3 Model

The way our model takes some file or snippet of source code as input and produces an output for a supervised learning task is sketched in Figure 1. In more detail, the model performs these four steps:

1. Parse the source code into an Abstract Syntax Tree.
2. Add edges of varying types (described in Appendix Table 4) to this syntax tree representing semantic information like data– and control– flow, in the spirit of [2]. Also add the reversed version

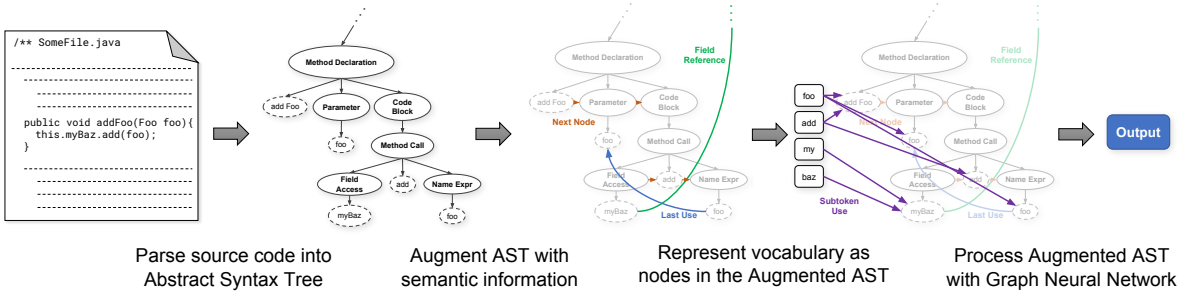

Figure 1: Our model's procedure for supervised learning on source code.

of all edges (with their own edge type) to permit messages to be sent in both directions between connected nodes in the graph. This results in a directed multigraph we call an Augmented AST.

3. Further augment the Augmented AST by adding a node for each vocabulary word encountered in the code. Each such "vocab node" is then connected with an edge to all variables whose names contain its word. (Following previous work we consider a variable's name to be a sequence of separate words, based on `CamelCase` or `snake_case` conventions. This is depicted in Figure 1.)

4. Process the Augmented AST with a Gated Graph Neural Network. To do this we must first vectorize the nodes of the Augmented AST. We vectorize vocab nodes using a Character–Level Convolutional Neural Network (CharCNN) [34] embedding of the node's word. We vectorize all other nodes with a learned embedding of their type. The type of a non–leaf node in the AST is the language construct it represents, e.g. Parameter, Method Declaration, etc. The type of a leaf node in the AST (i.e. a node representing a written token of code) is the name of the Java type of the token it contains, e.g. `int`, a user–defined class, etc.

   Processing by the GGNN results in hidden state vectors for every node in the Augmented AST, which are then used produce output in a task–specific way. (Described in the Experiments section.)

Our main contribution to previous works is the addition of Step 3. The combination of relational information from the vocab nodes' connections and lexical information from these nodes' CharCNN embeddings allows the model to, in principle, flexibly reason about words it never saw during training, but also recognize words it did. E.g. it could potentially see a class named "`getGuavamaticDictionary`" and a variable named "`guavamatic_dict`" and both (a) utilize the fact that the word "guavamatic" is common to both names despite having never seen this word before, and (b) exploit learned representations for words like "get", "dictionary", and "dict" that it has seen during training.

4 Experiments

Code to reproduce all experiments is available online.[1] [2]

We compared our model to two other models on two supervised tasks: a Fill–In–The–Blank task and a Variable–Naming task.

The models to which we compare differ from ours in how they treat the words they encounter in the code. The model referred to below as "Fixed Vocab" is very similar to the model from [2]. It follows

[1] https://github.com/mwcvitkovic/Deep_Learning_On_Code_With_A_Graph_Vocabulary–Code_Preprocessor
[2] https://github.com/mwcvitkovic/Deep_Learning_On_Code_With_A_Graph_Vocabulary

steps 1 and 2 from the steps in the Method section above, but it skips step 3, and in step 4 it uses a fixed vocabulary embedding to vectorize names (by looking up the embedding for the name's constituent words and taking their mean). The "CharCNN Only" model operates identically to this Fixed Vocab model, except it uses a CharCNN to vectorize the names of variables. The "Graph Vocab" model is our model that operates as described in the Method section.

For each task, we also compared the effects of augmenting the AST with extra edges (step 2 from the Method section). We list the performance of each model trained with the plain AST and the Augmented AST.

4.1 Data and Model Details

We randomly selected 18 of the 100 most popular Java repos from the Maven repository[3] to serve as training data. (See Appendix A for the list.) Together these repositories contain about 500,000 non–empty, non–comment lines of code. We randomly chose 3 of these repositories to sequester as an "entirely unseen repos" test set. We then separated out 15% of the files in the remaining 15 repositories to serve as our "unseen files from seen repos" test set. The remaining files served as our training set, from which we separated 15% of the datapoints to act as a validation set.

We used the open–source Javaparser[4] library to generate ASTs of our source code, and then used home–built code to augment the ASTs with the edges described in Appendix Table 4. We used MXNet as our deep learning framework, relying heavily on sparse operations for the GGNN implementation. All hidden states in the GGNN contained 64 units; all GGNNs ran for 8 rounds of message passing; all models were optimized using the Adam optimizer. [15]

4.2 The Fill–In–The–Blank Task

In this task we randomly selected a single usage of a variable in some source code, replaced it with a `<FILL-IN-THE-BLANK>` token, and then asked the model to predict what variable should have been there. An example instance is shown in Appendix Figure 2. We only selected variable usages where the variable is used somewhere else in the code, so the models could indicate their choice by neural attention. All models computed the attention weighting y_i for each variable i as per [19]:

$$y_i = \sigma\left(f_1(h_v^T, h_v^0)\right) \odot f_2(h_v^T),$$

where the fs are MLPs, h_v^t is the hidden state of node v after t GGNN message passing iterations, σ is the sigmoid function, and $\odot$ is elementwise multiplication. The models were trained using a binary cross entropy loss computed separately for each attended node. Performance is reported in Table 1.

4.3 The Variable Naming Task

In this task we replaced all usages of a name of a particular variable in the code with the text "`<NAME-ME>`", and asked the model to produce the correct name for this variable (in the form of the sequence of words that compose the name). An example instance is shown in Appendix Figure 3.

To produce a name from the output of the GGNN, our models followed [2] by taking the mean of the hidden states of the `<NAME-ME>` nodes and passing this as the initial hidden state to a 1–layer Gated Recurrent Unit (GRU) RNN [9]. To convert the hidden state output of this GRU into variable name

[3] https://mvnrepository.com/

[4] https://javaparser.org/

Table 1: Accuracy on the Fill–In–The–Blank task. A correct prediction is one in which the Augmented AST node that received the maximum attention weighting by the model contained the variable that was originally in the `<FILL-IN-THE-BLANK>` spot.

		Fixed Vocab	CharCNN Only	Graph Vocab (ours)
Unseen files from seen repos	AST	0.58	0.60	0.89
	Augmented AST	0.80	0.90	**0.97**
Entirely unseen repos	AST	0.36	0.48	0.80
	Augmented AST	0.59	0.84	**0.92**

predictions, the Fixed Vocab and CharCNN Only models passed the hidden states through a linear layer mapping to indices of words in their fixed vocabularies (i.e. a traditional decoder output for NLP). In contrast, the Graph Vocab model used the hidden state outputs of the GRU to compute dot–product attention with the hidden states of its vocab nodes and, as per [24], with a sentinel vector that lets the model fall back to a fixed vocabulary to produce words that are were visible in the graph. The models were trained by cross entropy loss over the sequence of words in the name. Performance is reported in Table 2.

Table 2: Performance on the Variable Naming task. Entries in this table are of the form "accuracy (edit distance)". A correct output is exact reproduction of the full name of the obfuscated variable. The edit distance is the mean of the character–wise Levenshtein distance between the produced name and the real name.

		Fixed Vocab	CharCNN Only	Graph Vocab (ours)
Unseen files from seen repos	AST	0.23 (7.22)	0.22 (8.67)	0.49 (3.87)
	Augmented AST	0.19 (7.64)	0.20 (7.46)	**0.53 (3.68)**
Entirely unseen repos	AST	0.05 (8.66)	0.06 (8.82)	0.38 (4.81)
	Augmented AST	0.04 (8.34)	0.06 (8.16)	**0.41 (4.28)**

5 Discussion

As can be seen in Tables 1 and 2, our Graph Vocab model outperforms the other models tested, and does comparatively well at maintaining accuracy between the seen and unseen test repos on the Variable Naming task.

The results reported herein are part of a continuing project, and we are pursuing several improvements to the model described above. In particular, we hope to add information about the word ordering in names and to improve how we vectorize information regarding a variable's data type. Also, there are many other types of Graph Neural Networks that could be used in place of the GGNN that are worth evaluating.

6 Acknowledgements

Many thanks to Miltos Allamanis and Hyokun Yun for their advice and useful conversations.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu & Charles Sutton (2017): *A Survey of Machine Learning for Big Code and Naturalness*. *arXiv:1709.06182 [cs]*. Available at `https://arxiv.org/abs/1709.06182`.

[2] Miltiadis Allamanis, Marc Brockschmidt & Mahmoud Khademi (2018): *Learning to Represent Programs with Graphs*. In: *International Conference on Learning Representations*. Available at `https://openreview.net/forum?id=BJOFETxR-`.

[3] Matthew Amodio, Swarat Chaudhuri & Thomas Reps (2017): *Neural Attribute Machines for Program Generation*. *arXiv:1705.09231 [cs]*. Available at `http://arxiv.org/abs/1705.09231`.

[4] Forough Arabshahi, Sameer Singh & Animashree Anandkumar (2018): *Combining Symbolic Expressions and Black-box Function Evaluations for Training Neural Programs*. In: *International Conference on Learning Representations*. Available at `https://openreview.net/forum?id=Hksj2WWAW`.

[5] Trapit Bansal, Arvind Neelakantan & Andrew McCallum: *RelNet: End-to-End Modeling of Entities & Relations*. *arXiv:1706.07179 [cs]*. Available at `http://arxiv.org/abs/1706.07179`.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin & Tomas Mikolov (2017): *Enriching Word Vectors with Subword Information*. *TACL* 5, pp. 135–146.

[7] Kwonsoo Chae, Hakjoo Oh, Kihong Heo & Hongseok Yang (2017): *Automatically Generating Features for Learning Program Analysis Heuristics for C-like Languages*. *Proc. ACM Program. Lang.* 1(OOPSLA), pp. 101:1–101:25, doi:10.1145/3133925. Available at `http://doi.acm.org/10.1145/3133925`.

[8] Xinyun Chen, Chang Liu & Dawn Song (2018): *Tree-to-tree Neural Networks for Program Translation*. Available at `https://openreview.net/forum?id=rkxY-sl0W`.

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk & Yoshua Bengio (2014): *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, pp. 1724–1734, doi:10.3115/v1/D14-1179. Available at `http://www.aclweb.org/anthology/D14-1179`.

[10] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals & George E. Dahl (2017): *Neural Message Passing for Quantum Chemistry*. In Doina Precup & Yee Whye Teh, editors: *Proceedings of the 34th International Conference on Machine Learning*, *Proceedings of Machine Learning Research* 70, PMLR, International Convention Centre, Sydney, Australia, pp. 1263–1272. Available at `http://proceedings.mlr.press/v70/gilmer17a.html`.

[11] C. Goller & A. Kuchler (1996): *Learning task-dependent distributed representations by backpropagation through structure*. In: *Neural Networks, 1996., IEEE International Conference on*, 1, pp. 347–352 vol.1, doi:10.1109/ICNN.1996.548916.

[12] L. Jiang, G. Misherghi, Z. Su & S. Glondu (2007): *DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*. In: *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, doi:10.1109/ICSE.2007.30.

[13] Daniel D. Johnson (2017): *Learning Graphical State Transitions*. In: *International Conference on Learning Representations*. Available at `https://openreview.net/forum?id=HJ0NvFzxl`.

[14] Yoon Kim, Yacine Jernite, David Sontag & Alexander M. Rush (2016): *Character-aware Neural Language Models*. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, AAAI Press, pp. 2741–2749. Available at `http://dl.acm.org/citation.cfm?id=3016100.3016285`.

[15] Diederik P. Kingma & Jimmy Ba (2015): *Adam: A Method for Stochastic Optimization*. In: *International Conference on Learning Representations*. Available at `https://arxiv.org/abs/1412.6980`.

[16] Sosuke Kobayashi, Ran Tian, Naoaki Okazaki & Kentaro Inui (2016): *Dynamic Entity Representation with Max-pooling Improves Machine Reading*. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Association for

Computational Linguistics, San Diego, California. Available at http://www.aclweb.org/anthology/N16-1099.

[17] Jens Krinke (2001): *Identifying Similar Code with Program Dependence Graphs*. In: *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, IEEE Computer Society, Washington, DC, USA, pp. 301–. Available at http://dl.acm.org/citation.cfm?id=832308.837142.

[18] Jian Li, Yue Wang, Irwin King & Michael R. Lyu (2017): *Code Completion with Neural Attention and Pointer Networks*. *arXiv:1711.09573 [cs]*. Available at http://arxiv.org/abs/1711.09573.

[19] Yujia Li, Daniel Tarlow, Marc Brockschmidt & Richard Zemel (2016): *Gated Graph Sequence Neural Networks*. In: *International Conference on Learning Representations*. Available at https://arxiv.org/abs/1511.05493.

[20] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez & Dawn Xiaodong Song (2017): *Neural Code Completion*. Available at https://openreview.net/forum?id=rJbPBt9lg&noteId=rJbPBt9lg.

[21] Zhengdong Lu, Haotian Cui, Xianggen Liu, Yukun Yan & Daqi Zheng (2017): *Object-oriented Neural Programming (OONP) for Document Understanding*. *arXiv:1709.08853 [cs]*. Available at http://arxiv.org/abs/1709.08853. ArXiv: 1709.08853.

[22] Minh-Thang Luong & Christopher D. Manning (2016): *Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models*. *arXiv:1604.00788 [cs]*. Available at http://arxiv.org/abs/1604.00788.

[23] Chris J. Maddison & Daniel Tarlow (2014): *Structured Generative Models of Natural Source Code*, pp. II–649–II–657. Available at http://dl.acm.org/citation.cfm?id=3044805.3044965.

[24] Stephen Merity, Caiming Xiong, James Bradbury & Richard Socher (2017): *Pointer Sentinel Mixture Models*. In: *International Conference on Learning Representations*. Available at https://openreview.net/pdf?id=Byj72udxe.

[25] Trang Pham, Truyen Tran & Svetha Venkatesh: *Graph Memory Networks for Molecular Activity Prediction*. *arXiv:1801.02622 [cs]*. Available at http://arxiv.org/abs/1801.02622.

[26] Maxim Rabinovich, Mitchell Stern & 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 (Volume 1: Long Papers)*, Association for Computational Linguistics, pp. 1139–1149, doi:10.18653/v1/P17-1105. Available at http://www.aclweb.org/anthology/P17-1105.

[27] Veselin Raychev, Martin Vechev & Andreas Krause (2015): *Predicting Program Properties from "Big Code"*. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, ACM, New York, NY, USA, pp. 111–124, doi:10.1145/2676726.2677009. Available at http://doi.acm.org/10.1145/2676726.2677009.

[28] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov & Max Welling (2017): *Modeling Relational Data with Graph Convolutional Networks*. *arXiv:1703.06103 [cs, stat]*. Available at http://arxiv.org/abs/1703.06103.

[29] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng & Christopher Potts (2013): *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank*. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Seattle, Washington, USA, pp. 1631–1642. Available at http://www.aclweb.org/anthology/D13-1170.

[30] Oriol Vinyals, Meire Fortunato & Navdeep Jaitly (2015): *Pointer Networks*. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama & R. Garnett, editors: *Advances in Neural Information Processing Systems 28*, Curran Associates, Inc., pp. 2692–2700. Available at http://papers.nips.cc/paper/5866-pointer-networks.pdf.

[31] M. White, M. Tufano, C. Vendome & D. Poshyvanyk (2016): *Deep learning code fragments for code clone detection*. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 87–98. Available at http://www.cs.wm.edu/~mtufano/publications/C5.pdf.

[32] Pengcheng Yin & 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 (Volume 1: Long Papers)*, Association for Computational Linguistics, pp. 440–450, doi:10.18653/v1/P17-1041. Available at http://www.aclweb.org/anthology/P17-1041.

[33] Wojciech Zaremba, Karol Kurach & Rob Fergus (2014): *Learning to Discover Efficient Mathematical Identities*. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence & K. Q. Weinberger, editors: *Advances in Neural Information Processing Systems 27*, Curran Associates, Inc., pp. 1278–1286. Available at http://papers.nips.cc/paper/5350-learning-to-discover-efficient-mathematical-identities.pdf.

[34] Xiang Zhang, Junbo Zhao & Yann LeCun (2015): *Character-level Convolutional Networks for Text Classification*. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama & R. Garnett, editors: *Advances in Neural Information Processing Systems 28*, Curran Associates, Inc., pp. 649–657. Available at http://papers.nips.cc/paper/5782-character-level-convolutional-networks-for-text-classification.pdf.

7 Appendix

Figures 2 and 3 show, respectively, an example instance of the Fill–In–The–Blank and Variable Naming tasks. Table 3 lists the repositories used to in our dataset. Table 4 lists the edges added to the AST to create the Augmented AST.

```
public static boolean isPrime(int n) {
        if (n < 2) {
                return false;
        }

        for (int p : SmallPrimes.PRIMES) {
                if (0 == (n % p)) {
                        return n == p;
                }
        }
        return SmallPrimes.millerRabinPrimeTest(n);
}
```

Figure 2: Example instance of the Fill–In–The–Blank task. This particular Fill–In–The–Blank instance is created by replacing the red–highlighted usage of the variable "n" with the special token <FILL-IN-THE-BLANK>. The model then processes the Java file containing this snippet according to Figure 1 to produce an Augmented AST. This Augmented AST is too large to show here, but the model indicates which variable was originally present in the <FILL-IN-THE-BLANK> slot by attending to nodes in this Augmented AST. So if the model places maximal attention on any of the nodes representing the green–highlighted variables, this is a correct output. If maximal attention is placed on any other node in the Augmented AST it is an incorrect output.

```
String header = message == null ? "" : message + ": ";

int expectedsLength = assertArraysAreSameLength(expecteds,
        actuals, header);

for (int i = 0; i < expectedsLength; i++) {
    Object expected = Array.get(expecteds, i);
    Object actual = Array.get(actuals, i);

    if (isArray(expected) && isArray(actual)) {
        try {
            arrayEquals(message, expected, actual);
        } catch (ArrayComparisonFailure e) {
            e.addDimension(i);
            throw e;
        }
    } else {
        try {
            assertElementsEqual(expected, actual);
        } catch (AssertionError e) {
            throw new ArrayComparisonFailure(header, e, i);
        }
    }
}
```

Figure 3: Example instance of the Variable Naming task. This particular Variable Naming instance is created by replacing all uses of the name of the "expectedsLength" variable, shown highlighted in green, with "<NAME-ME>". The model then processes the Java file containing this snippet according to Figure 1 to produce an Augmented AST. It then generates a sequence of tokens as its output in an attempt to generate the name of the obfuscated variable; in this case the correct output would be "['expecteds', 'length', '<EOS>']".

Table 3: Repositories used in experiments. All were taken from the Maven repository (https://mvnrepository.com/). Entries are in the form "group/repository name/version".

Seen Repos
com.fasterxml.jackson.core/jackson-core/2.9.5
com.h2database/h2/1.4.195
javax.enterprise/cdi-api/2.0
junit/junit/4.12
mysql/mysql-connector-java/6.0.6
org.apache.commons/commons-collections4/4.1
org.apache.commons/commons-math3/3.6.1
org.apache.commons/commons-pool2/2.5.0
org.apache.maven/maven-project/2.2.1
org.codehaus.plexus/plexus-utils/3.1.0
org.eclipse.jetty/jetty-server/9.4.9.v20180320
org.reflections/reflections/0.9.11
org.scalacheck/scalacheck_2.12/1.13.5
org.slf4j/slf4j-api/1.7.25
org.slf4j/slf4j-log4j12/1.7.25
Unseen Repos
org.javassist/javassist/3.22.0-GA
joda-time/joda-time/2.9.9
org.mockito/mockito-core/2.17.0

Table 4: Edge types used in Augmented ASTs. The initial AST is constructed using the `AST` and `NEXT_TOKEN` edges, and then the remaining edges are added. The reversed version of every edge is also added as its own type (e.g. `reverse_AST`, `reverse_LAST_READ`) to let the GGNN message passing occur in both directions.

Edge Name	Description
`AST`	The edges used to construct the original AST.
`NEXT_TOKEN`	Edges added to the original AST that specify the left–to–right ordering of the children of a node in the AST. These edges are necessary since ASTs have ordered children, but we are representing the AST as a directed graph.
`COMPUTED_FROM`	Connects a node representing a variable on the left of an equality to those on the right. (E.g. edges from `y` to `x` and `z` to `x` in `x = y + z`.) The same as in [2].
`LAST_READ`	Connects a node representing a usage of a variable to all nodes in the AST at which that variable's value could have been last read from memory. The same as in [2].
`LAST_WRITE`	Connects a node representing a usage of a variable to all nodes in the AST at which that variable's value could have been last written to memory. The same as in [2].
`RETURNS_TO`	Points a node in a return statement to the node containing the return type of the method. (E.g. `x` in `return x` gets an edge pointing to `int` in `public static int getX(x)`.)
`LAST_SCOPE_USE`	Connects a node representing a variable to the node representing the last time this variable's name was used in the text of the code (i.e. capturing information about the text, not the control flow), but only within lexical scope. This edge exists to try and give the non–Graph–Vocab models as much lexical information as possible to make them as comparable with the Graph Vocab model.
`LAST_FIELD_LEX`	Connects a field access (e.g. `this.whatever` or `Foo.whatever`) node to the last use of this.whatever (or to the variable's initialization, if it's the first use). This is not lexical–scope aware (and, in fact, can't be in Java, in general).
`FIELD`	Points each node representing a field access (e.g. `this.whatever`) to the node where that field was declared.
`SUBTOKEN`	Points vocab nodes to nodes representing variables in which the vocab word was used in the variable's name.