A Neural Decompiler for Java Classes

Anonymous ACL submission

Abstract

We propose a novel approach using neural machine translation to automatically decompile entire Java classes. Our method relies only on {source code, bytecode} pairs of Java methods and does not require any additional domain knowledge of the target language. To overcome the token length limitations of current Transformer models, we partition class bytecode into methods, generate Java code for each method, and then reassemble all outputs into a final class. Our neural decompiler is able to generate more human-readable output (measured by CodeBLEU) than existing software-based decompilers while achieving slightly lower pass rates on fuzz tests. We will release our source code, dataset collection code, and pretrained Java class decompiler model to aid in development of more robust neural machine translators.

1 Introduction

Decompilation is the process of converting a binary machine language into a corresponding high-level language source code. This technique has numerous applications in fields such as rewriting legacy code, malware analysis, and software vulnerability repair. Unfortunately, existing software-based decompilers are time-consuming to develop and can generate source code that is hard for humans to understand (Hosseini and Dolan-Gavitt, 2022).

Neural Machine Translation (NMT) methods have been recently proposed as an alternative to conventional software solutions to translate between programming languages (e.g., C# to Java) (Wang et al., 2021; Szafraniec et al., 2022). NMT approaches have also been applied to program decompilation, where the source language is a compiled assembly/bytecode representation generated by a compiler and the target language is the original programming language.

The majority of NMT approaches focus on translating a single function with no side effects. We speculate this contraint is due in large part to the limited source and target lengths for Transformer-based translation models. For instance, CodeT5 (Wang et al., 2021) uses source and target sequence lengths of 512 and 256, respectively. This problem is exacerbated when the source sequence is an assembly/bytecode representation that can require 2-8x more tokens than their programming language counterpart.

In this paper, we focus on the task of translating the Java bytecode of an entire class file to the original Java source code. This problem is significantly more challenging than translating a single function for multiple reasons. First, a class can contain tens of methods that, when tokenized, may exceed a default 512 token limit by 10-20x. Often fields/methods defined earlier in a class are used in the implementation of other methods, making correct decompilation challenging if they are no longer in the context window. Similarly, imported packages, generally defined at the top of the class, are also used throughout the file. Second, a Java class often contains mutable member variables (fields) that can be used in any method. Finally, there are many language-specific features that generate more rarely occurring patterns of bytecode (e.g., exceptions, static/final variables, multiple constructors).

To address these challenges, we have developed a bytecode partitioning strategy, used during both training and inference, that contains all necessary information for correct decompilation while fitting within a relatively constrained sequence length. Specifically, we construct custom header format that contains type information on all methods/-fields in the class as well as all imported package names. Next, we break the class bytecode on method boundaries (including constructors) and prepend the header to each method. After generating the corresponding Java code for each method, we assemble the Java class by simple concatenating each method. For large classes with many methods, we perform the method-level generation in batches

to speed up the decompilation process. Surprisingly, due to the parallelized nature of Transformers, our approach achieves similar decompilation runtimes (using an A100 GPU) as conventional software-based decompilers running on CPUs. Finally, using a fuzz-testing framework, we can test the quality of the decompiled code. Generally, we find our NMT-based approach achieves a slightly lower test pass rate but leads to higher quality code using CodeBLEU (Ren et al., 2020) compared to state-of-the-art software-based decompilers.

2 Related Work

2.1 Software-based Decompilers

Decompilation is the process of converting binary/assembly/bytecode generated by a compiler back to the original high-level language. Decompilation is often more difficult than compilation because much of the information in source file, such as variable names and original control flow, has been removed. Many techniques/heuristics have been developed over time to estimate the original source file with absence of complete information (Cifuentes and Gough, 1995).

We compare our approach against several open-source Java decompilers that have been in development over a long period of time (Benfield, 2022; skylot, 2022; mstrobel, 2022; Storyyeller, 2022; fesh0r, 2022). Harrand et al. provide a detailed analysis the quality of the source code generated by these decompilers (Harrand et al., 2019). For simple classes, all decompilers are able to provide accurate and readable Java. However, for more complicated class methods (e.g., deeply nested code with complex control flow), they can generate code that fails to compile. More often, they will generated semantically correct code but written in a non-intuitive way.

2.2 NMT-based Decompilers

Katz et al. framed LLVM-IR (intermediate representation) to C decompilation as a translation problem using a recurrent neural network (Katz et al., 2018). This work constrained the problem to short code snippets (max of 112 binary tokens and 88 source code tokens). DIRE focused on the sub-problem of generating good names for identifiers for x86-64 binary to C decompilation (Lacomis et al., 2019). Coda developed an instruction-aware AST (for C programs) to restrict invalid token generation of an LSTM model (Fu et al., 2019).

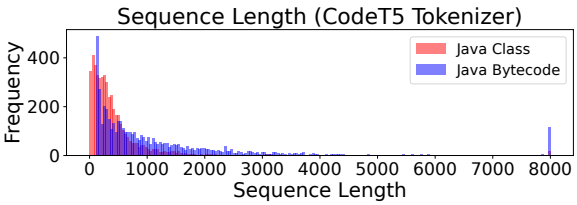

Figure 1: Token sequence length (using the CodeT5 Tokenizer) for 5000 Java classes (in red) and their corresponding bytecode representation (in blue). Sequences longer than 8000 were truncated in the figure.

BTC developed a language agnostic decompiler to generate functions from assembly to many source languages (C/Go/Fortran/OCaml) using a single model (Hosseini and Dolan-Gavitt, 2022).

Compared to this prior work, we believe we are the first to tackle full Java class decompilation where both source and bytecode token lengths can be significantly longer than a 512 token limit (up to 10k tokens per class).

3 Constructing a Java Bytecode Dataset

We extract Java classes from Github repositories indexed by Google BigQuery[1]. In order to generate bytecode, we must be able to compile these files with minimal configuration overhead. Therefore, we discard files with 3rd party imports (anything not starting with `import java.*`). Additionally, we discard files containing multiple classes.

After these preprocessing steps, we split the java classes into a training and testing set with 150k classes and 20k classes, respectively. For each class, we used the Java 8 compiler to generate bytecode for each class. This bytecode was then disassembled using Krakatau (Storyyeller, 2022) to achieve a human-readable bytecode representation. We use this disassembled bytecode representation as input to our NMT model. Figure 1 shows the sequence length of Java classes and disassembled bytecode representations after being tokenized with the CodeT5 tokenizer (Wang et al., 2021). For any given Java class, the bytecode is often 3-4x longer.

Following the same approach as (Roziere et al., 2021), we generate unit tests for each Java class via fuzz testing using EvoSuite (Fraser and Arcuri, 2011) and keep test with a mutation score larger than 90%. For any decompiler (either software-based or NMT-based), use these tests to validate that the decompiled Java class performs logically

[1] https://console.cloud.google.com/marketplace/details/github/github-repos

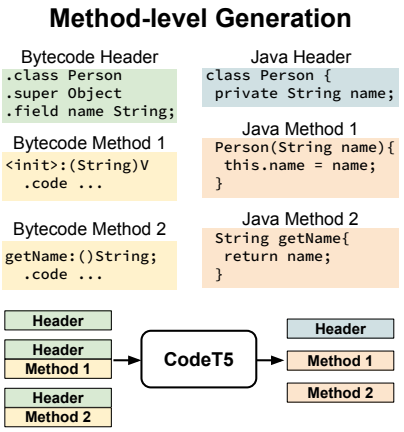

Figure 2: (top left) Disassembled bytecode for a Java class with one constructor and one method. (top right) The target Java class used to generate the bytecode. (bottom) Each bytecode method is decompiled separately by our finetuned CodeT5 model.

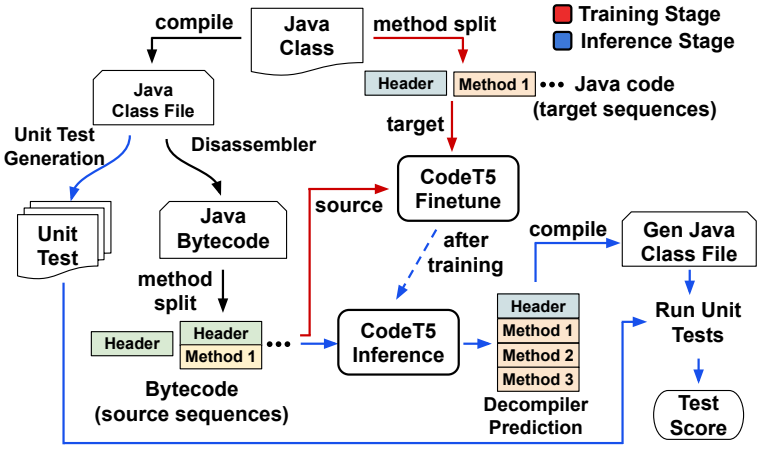

Figure 3: An overview of our training and inference methodology. During training (red arrows), pairs of Java code methods and compiled bytecode methods used to finetune a CodeT5 model. During inference (blue arrows), the trained CodeT5 model is used to generate a decompiler prediction on bytecode methods. The functionality of the generated class is evaluated using unit tests obtained from the ground-truth Java class.

the same as the original ground-truth class. We will discuss this evaluation process in more details in the next section.

4 Java Class Neural Decompiler

4.1 Method-level Generation

Based on the sequences lengths shown in Figure 1, simply finetuning an existing code language model, such as CodeT5, will fail to achieve reasonable performance as many Java classes are significant longer than the sequence lengths typically used. Even when using much longer sequence lengths then CodeT5 (i.e., a maximum source sequence length of 2048 and a maximum target sequence of 512), 29.1% of samples fail to fit.

We propose a method-level generation approach to address this sequence length issue in Java class decompilation. This approach breaks down the problem into multiple sub-problems by decompiling a class header (imports and fields) and decompiling each class method independently. Figure 2 provides an overview of the approach. The bytecode header is prepended to each method so that it has access to necessary context such as the imports, type signatures for other methods in the class, and types of fields. Additionally, the bytecode header is treated as a sample by itself, in order to generate the corresponding header portion of the Java program (i.e., imports and field declarations). Once decompilation has been performed for each method, we simply concatenate them together, starting with the header, in order to generate our final decompiled Java class. See Appendix C for more information the structure of these headers.

Decompilation via method splitting has two advantages compared to decompiling the entire class bytecode. As mentioned earlier, the main benefit is reducing both the source and target sequence lengths, as generation is now performed per method. The bytecode header that is prepended to each method before generation is typically around 50-100 tokens and provides all type information for methods/fields in the class without giving any additional (likely unnecessary) implementation details. The second benefit is that each method can be generated (i.e., decompiled) in parallel via batching which significantly reduces the generation time for classes with many methods.

4.2 Training and Evaluation Methodology

Figure 3 presents the training and evaluation methodology for our method-level Java decompiler. First, {bytecode, source code} pairs of Java methods are collected to serve as the training set for a CodeT5 model. For each bytecode method, the bytecode header is prepended as shown in Figure 2. After training, Java class files from a test set are similarly disassembled and converted into a set of source sequence before being passed to the CodeT5 model. The model generates a decompiled method for each of the source sequences, which we then concatenate to get our compiler prediction. The

ground-truth Java class is passed through Unit Test Generator (Evosuite) to generate tests for all methods in the class. A test score for the decompiled class is then computed by running it against the unit tests.

5 Training and Evaluation Results

5.1 Training

We use the same finetuning methodology as in CodeT5. We start a pre-trained CodeT5-base model (220M parameters) that was fine-tuned on C# to Java code translation task. We trained on 150k Java classes for 3 epochs. We used source and target sequence lengths of 2048 and 512, respectively, to handle longer methods. Training was performed on 4 A100 GPUs and took 20 hours to complete. See Appendix A for more details.

5.2 Evaluation Metrics

We use unit tests, generated by Evosuite, to judge whether the decompiled class is functionally the same as the ground-truth Java class. The number of generated tests is determined by the complexity of behavior in a given method. Our decompiled class is finally run against these ground-truth unit tests to see if it matches the functionality of the ground-truth Java class. We consider a decompiled class to pass only if all of the unit tests pass.

Additionally, we use CodeBLEU (Ren et al., 2020) to measure the code similarity of decompiled class to that of the ground-truth Java class. Compared to BLEU (Papineni et al., 2002) which matches n-grams between two sequences, CodeBLEU add three additional components: weighted n-gram matching the emphasizes keywords and variable names, syntactic AST match that compares the syntax trees, and dataflow match that compares variable dataflow of the sequences.

5.3 Results

We compare our CodeT5 decompiler against 5 software-based decompilers on 2k Java classes, which follow a sequence length distribution similar to Figure 1. Of the software-based decompiler, CFR performs the best with a pass rate of 98.5%, a CodeBLEU score of 0.49, and an average decompile time 0.29 seconds. All but one decompiler (Krakatau) achieve a pass rate above 97%. By comparison, our CodeT5 decompiler achieves a pass rate of 93.4% (matching Krakatau) with an average decompile time of 1.27 seconds (using a single A100 GPU). Note that generations were performed in half precision (fp16) to reduce runtime as there was no observable difference in performance compared to full precision (full generation settings in Appendix B). Common failure modes of our decompiler are discussed in Appendix E. Many of these failure modes are tied to missing information in the bytecode header, making it difficult for the model to generate valid Java code.

Table 1: Decompiler evaluation. Pass rate is % of decompiled classes that pass all tests. CodeBLEU (total) equally weights ng (n-gram), wng (weighted n-gram), sm (syntax match), and dm (dataflow match). Time is the average decompilation time across all test samples.

Decompiler	Pass Rate	CodeBLEU					Time
		total	ng	wng	sm	dm	
Procyon	98.1%	0.48	**0.24**	0.32	0.63	**0.74**	0.44s
CFR	**98.5%**	0.49	0.23	0.30	0.76	0.65	0.29s
JADX	97.1%	0.47	0.23	0.30	0.75	0.60	1.16s
Fernflower	97.7%	0.47	0.20	0.29	0.76	0.63	0.30s
Krakatau	93.4%	0.38	0.16	0.22	0.64	0.49	**0.22s**
Ours	93.4%	**0.53**	0.23	**0.37**	**0.86**	0.66	1.27s

Interestingly, our decompiler achieves the highest CodeBLEU score by a significant margin (0.53 compared to 0.49 for best software decompiler). Looking at the component breakdown, we see that our decompiler outperforms the other decoders in terms of weighted n-gram match and syntax match. We believe that generating more human-readable results is the biggest advantage of our approach. For instance, the model is able to deduce likely variable names for local variables based on other names in the class and the structure of the program. Similarly, it is able to deduce program structure that was optimized away during compilation. Conventional decompilers require heuristics to make reasonable guesses about these types of issues.

6 Conclusions

In this work, we describe a simple methodology for training a encoder-decoder model (CodeT5) to decompile entire Java classes. Using a common header, we show that method-level generation is a reasonable way to overcome the sequence length limitations of current Transformer architectures, leading to an approach that can support long classes (e.g., 8k tokens) with many methods. Compared to existing software-based decompilers, our approach achieves a slightly lower pass rate, but generates code with a higher CodeBLEU score. We hope our method-splitting approach will lead to additional work on full-program translation/decompilation.

7 Ethical Considerations

The field of decompilation, and specifically the use of neural machine translation (NMT) models for decompilation, raises a number of ethical considerations. In this section, we will discuss some of the key concerns that arise in this context.

7.1 Generation of Nefarious or Invalid Code

One unique concern with NMT-based decompilation is that it may generate code that is invalid or malicious in ways that differ from conventional software-based decompilers. For example, a decompiler might produce code that appears syntactically correct, but that has unintended or malicious side effects when executed. This could be a result of the model failing to accurately understand the original code, or it could be due to the model being intentionally fed specific bytecode samples for the purpose of generating malicious code.

To mitigate this risk, it is important to make these types of issues known and to carefully evaluate the code generated by NMT-based decompilers and to use appropriate testing/validation techniques.

7.2 Software Reverse Engineering

Another ethical concern with NMT-based decompilation is the potential for it to be used for software reverse engineering. Reverse engineering is the process of taking apart a piece of software in order to understand how it works, or to identify vulnerabilities or other weaknesses. In some cases, reverse engineering may be done for legitimate purposes, such as to identify and fix security vulnerabilities or to develop compatibility or interoperability solutions. However, in other cases, it may be used for nefarious purposes, such as to steal intellectual property or to create competing software products.

While reverse engineering is possible using conventional software-based decompilers, the improved syntactic structure and clearer variables names of NMT-based decompilers like our approach may lower the barrier of entry for many programmers. This could lead to an increase in the number of individuals and organizations engaging in software reverse engineering, which could pose a threat to the intellectual property and competitive advantage of software companies.

To address these ethical concerns, it may be necessary to put measures in place to restrict the use of NMT-based decompilers to only those with legitimate purposes. This could include the implementation of licensing or access controls, as well as educational campaigns to raise awareness about the potential consequences of software reverse engineering. It may also be necessary to address any legal or regulatory issues surrounding the use of these tools, such as clarifying the boundaries of fair use and protecting the rights of software developers. Ultimately, the responsible use of NMT-based decompilers will require a balance between the benefits they offer and the potential risks they pose.

7.3 Security and Privacy

Finally, there are also potential security and privacy concerns related to NMT-based decompilation. Decompiling software may reveal sensitive information, such as hardcoded passwords or keys, which could be exploited by malicious actors. In addition, decompiling software may reveal vulnerabilities or weaknesses in the code, which could be exploited to gain unauthorized access or to disrupt the software's functionality. Again, while this is already possible with conventional decompilers, as NMT-based decompilers improve the readability of code, it could become a larger risk.

7.4 Summary

In summary, the development and use of NMT-based decompilers raises a number of ethical concerns that should be carefully considered. These include the potential for the generation of nefarious or invalid code, the use of decompilers for software reverse engineering, intellectual property concerns, and issues related to security and privacy. While these concerns are not unique to NMT-based decompilers, the improved capabilities of these tools may make them more appealing to those with malicious intent. Therefore, it is important for researchers and practitioners in this field to carefully consider these ethical implications and to take steps to minimize potential negative consequences. This may include carefully controlling access to these tools, implementing safeguards to prevent the generation of invalid or malicious code, and working with legal and policy experts to ensure that these tools are used responsibly and in compliance with relevant laws and regulations.

References

L. Benfield. 2022. Cfr - yet another java decompiler. Last accessed 25 November 2022.

Cristina Cifuentes and K John Gough. 1995. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829.

fesh0r. 2022. Fernflower. Last accessed 25 November 2022.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32.

Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. 2019. The strengths and behavioral quirks of java bytecode decompilers. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 92–102. IEEE.

Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the c: Retargetable decompilation using neural machine translation.

Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356.

Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE.

mstrobel. 2022. procyon. Last accessed 25 November 2022.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*.

skylot. 2022. Jadx. Last accessed 25 November 2022.

Storyyeller. 2022. Krakatau. Last accessed 25 November 2022.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

A Training Settings

We used the HuggingFace Transformers summarize script[2] with the following settings. We made minimal modifications to the script to account for different key names for the source and target fields. We used a pretrained CodeT5-base model that was pretrained on C#-to-Java translation as its objective. Otherwise, we simply used the default learning rate, learning rate schedule, optimizer, weight decay, etc... We did not perform any hyperparameter search.

```
python run_summarization.py \
 --model_name_or_path
  <cs_java_codet5_base> \
 --tokenizer_name codet5-base \
 --do_train \
 --do_eval \
 --train_file <train.json> \
 --validation_file <test.json> \
 --output_dir <output_dir> \
 --overwrite_output_dir \
 --max_source_length 2048 \
 --max_target_length 512 \
 --per_device_train_batch_size=1 \
 --per_device_eval_batch_size=1 \
 --save_total_limit 1 \
 --predict_with_generate
```

[2] https://github.com/huggingface/transformers/tree/main/examples/pytorch/summarization

B Generation Settings

Similar to training, for generation we used the HuggingFace Transformers library. Specifically, we used the `generate` function with a batch size of 4, a `max_length` of 512. We did not use beam search, modify temperature/top-k/top-p, or use sampling. We performed a single generation per method and always used it. We did experiment with these settings, but found it had little impact on the generated output and never changed a failing test to a passing one.

For classes with more than 4 methods, we chunked the methods into many batches of 4, performed generation for each batch, and then combined all output to generate a complete Java class.

C Class Header Format

We add method signatures and imports to the bytecode header so that the CodeT5 decompiler model has access to all important context. Figure 4 provides an example of this header format for a class with one constructor and four methods. As a preprocessing step, we search through the entire bytecode file and record all defined methods and used third-party library functions. Then, we modify the header generated by Krakatau to include method signatures and imports.

D Code Generation Comparison

In this section, we show some qualitative comparisons for code generated by our model compared to the other decompiler. These samples are picked to point out interesting differences between NMT-based decompilers and software-based ones. In many cases, both types of decompilers may provide identical output.

Figure 5 shows an example `test` class decompiled using Procyon (left) and our CodeT5 model (right). Our model provides a perfect syntax match to the ground-truth model and also has a higher weighted n-gram match. Figure 6 provides another example for a `CoinChangingMinimumCoin` class. Our decompiler provides significantly more clear variable names (e.g., coins instead of array2). Additionally, Procyon adds the final keyword to most variable declarations. These do not appear in the original source code.

E Decompiler Failure Modes

In this section, we illustrate some common failure modes of our CodeT5 decompiler. Figure 7 compares the ground-truth Java class (the target) with our CodeT5 output for the `IntChPair` class. When using % modifiers like %s, our model will sometimes inject additional characters that do not belong. In this case, it add a unicode character. We are unsure what causes this, but suspect it may be related to the tokenizer and something in the bytecode represetation.

A second failure mode is related to the use of large constant values. Figure 8 compares the ground-truth to our output a `Hash` class. Here, our model uses an (incorrect) hexadecimal value instead of a decimal value. This type of failure has been frequently observed by others. LLMs are generally poor at reliably handling large numbers (especially doing conversions between bases or arithmetic).

Finally, a third (quite common) failure mode is shown in Figure 9. Here, our model does not initialize the LinkedHashMap during declaration. This will lead to an error later when the hashmap is used as it has not been initialized. This type of error accounts for around 30% of all failures in our test dataset. Likely, we are missing some important context related to fields initalized in this manner, so the model has no chance to learn how to perform this generation. We hope fixing this will lead to a significant improvement in our decompiler pass rate.

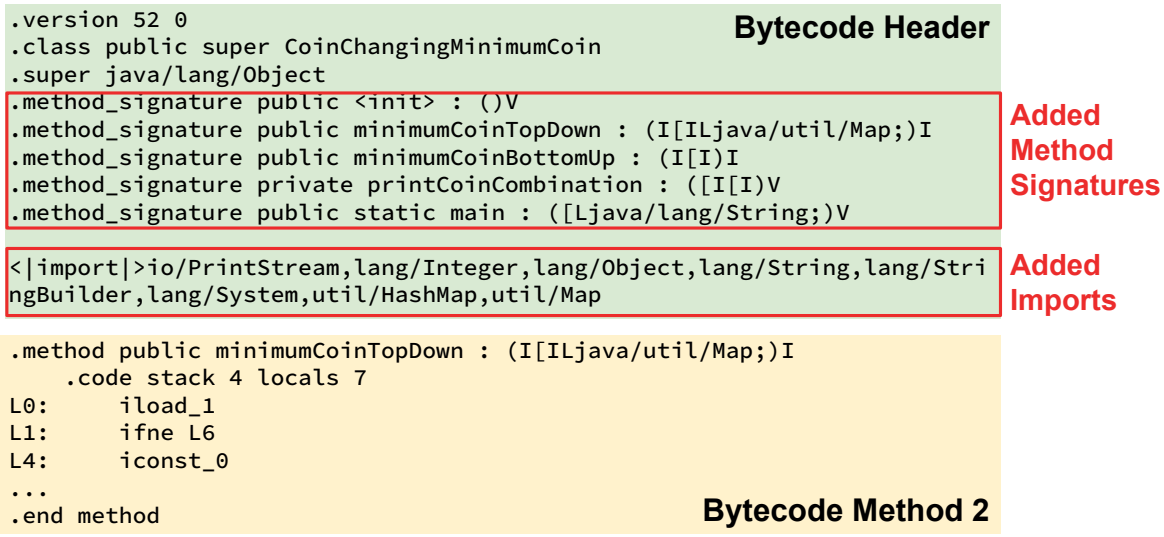

Figure 4: A bytecode header (top) is prepend to a bytecode method (bottom) before being passed to the CodeT5 model. If the header is passed into the model by itself, we use a <|header|> token to instruct the model to generate the corresponding Java header.

```java
import java.util.ArrayList;
public class test

{
   public static void main(final String[] array) {
       final int[][] array2 = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10
}, { 11, 12, 13, 14, 15 }, { 16, 17, 18, 19, 20 }, { 21, 22, 23, 24,
25 } };
       final ArrayList list = new ArrayList();
       final int n = 2;
       final int n2 = 2;
       final int n3 = 2;
       for (int i = 0; i < 5; ++i) {
           for (int j = 0; j < 5; ++j) {
               if ((i == n - n3 && n2 + n3 >= j && j >= n2 - n3) ||
(i == n + n3 && n2 + n3 >= j && j >= n2 - n3) || (j == n2 - n3 && n
+ n3 >= i && i >= n - n3) || (j == n2 + n3 && n + n3 >= i && i >= n
- n3)) {
                   list.add(array2[i][j]);
               }
           }
       }
       System.out.println(list);
   }
}
```

```java
import java.util.ArrayList;
public class test {

   public static void main(String[] args) {
       int[][] matrix = {
               {1, 2, 3, 4, 5},
               {6, 7, 8, 9, 10},
               {11, 12, 13, 14, 15},
               {16, 17, 18, 19, 20},
               {21, 22, 23, 24, 25}
       };
       ArrayList<Integer> list = new ArrayList<Integer>();
       int x = 2;
       int y = 2;
       int z = 2;
       for (int i = 0; i < 5; i++) {
           for (int j = 0; j < 5; j++) {
               if (i == x - z && y + z >= j && j < y - z
                       || i == x + z && y + z >= j && j < y - z
                       || j == y - z && x + z >= i && i < x - z) {
                   if (j == y + z && x + z >= i && i >= x - z) {
                       list.add(matrix[i][j]);
                   }
               }
           }
       }
       System.out.println(list);
   }
   public test() {}
}
```

Figure 5: (left) Procyon generation and (right) generation from our CodeT5 model.

```
import java.util.HashMap;
import java.util.Map;

public class CoinChangingMinimumCoin
{
   private void printCoinCombination(final int[] array,
final int[] array2) {
       if (array[array.length - 1] == -1) {
           System.out.print("No solution is possible");
           return;
       }
       int i = array.length - 1;
       System.out.print("Coins used to form total ");
       while (i != 0) {
           final int n = array[i];
           System.out.print(array2[n] + " ");
           i -= array2[n];
       }
       System.out.print("\n");
   }

   public static void main(final String[] array) {
       final int n = 13;
       final int[] array2 = { 7, 3, 2, 6 };
       final CoinChangingMinimumCoin
coinChangingMinimumCoin = new CoinChangingMinimumCoin();
       System.out.print(String.format("Bottom up and top
down result %s %s",
coinChangingMinimumCoin.minimumCoinBottomUp(n, array2),
coinChangingMinimumCoin.minimumCoinTopDown(n, array2, new
HashMap<Integer, Integer>())));

    }
}
```

```
import java.util.HashMap;
import java.util.Map;

public class CoinChangingMinimumCoin {
   private void printCoinCombination(int[] coins, int[]
cost) {
       if (coins[coins.length - 1] == -1) {
           System.out.print("No solution is possible");
           return;
       }

       int start = coins.length - 1;
       System.out.print("Coins used to form total ");
       while (start!= 0) {
           int cur = coins[start];
           System.out.print(cost[cur] + " ");
           start = start - cost[cur];
       }
       System.out.print("\n");
   }

   public static void main(String[] args) {
       int n = 13;
       int[] coins = {7, 3, 2, 6};
       CoinChangingMinimumCoin c = new
CoinChangingMinimumCoin();
       Map<Integer, Integer> map = new HashMap<>();
       int result = c.minimumCoinTopDown(n, coins, map);
       int bottomUp = c.minimumCoinBottomUp(n, coins);

       System.out.print(String.format("Bottom up and top
down result %s %s", bottomUp, result));
   }
}
```

Figure 6: (left) Procyon generation and (right) generation from our CodeT5 model.

```
public class IntChPair {
   int a;
   char b;

   public IntChPair(int a, char b) {
       this.a = a;
       this.b = b;
   }
   public int getA() {
       return a;
   }
   public void setA(int a) {
       this.a = a;
   }
   public char getB() {
       return b;
   }
   public void setB(char b) {
       this.b = b;
   }
   @Override
   public boolean equals(Object obj) {
       if (!(obj instanceof IntChPair)) return false;
       IntChPair o = (IntChPair) obj;
       return this.a == o.getA() && this.b == o.getB();
   }
   @Override
   public int hashCode() {
       return a ^ b;
   }
   @Override
   public String toString() {
       return String.format("(%s %s)", a, b);
   }
}
```

```
public class IntChPair {
   int a;
   char b;

   public IntChPair(int a, char b) {
       this.a = a;
       this.b = b;
   }
   public int getA() {
       return a;
   }
   public void setA(int a) {
       this.a = a;
   }
   public char getB() {
       return b;
   }
   public void setB(char b) {
       this.b = b;
   }
   @Override
   public boolean equals(Object o) {
       if (!(o instanceof IntChPair)) return false;
       IntChPair p = (IntChPair) o;
       return a == p.getA() && b == p.getB();
   }
   @Override
   public int hashCode() {
       return a ^ b;
   }
   @Override
   public String toString() {
       return String.format("(%s\u00A7%s)", a, b);
   }
}
```

Figure 7: (left) Ground-truth Java class and (right) generation from our CodeT5 model. Our model injects a Unicode character in the string format between each "%s".

```java
public class Hash {
   public static int computeHash(byte[] data) {
       final int p = 16777619;
       int hash = (int) 166136261;

       for (int i = 0; i < data.length; ++i)
         hash = (hash ^ data[i]) * p;

       hash += hash << 13;
       hash ^= hash >> 7;
       hash += hash << 13;
       hash ^= hash >> 7;
       hash += hash << 3;
       hash ^= hash >> 17;
       hash += hash << 5;
       return hash;
   }
}
```

```java
public class Hash {

   public static int computeHash(byte[] data) {
       final int HASH_SIZE = 166136261;
       int hash = 0x5bd1e995;
       for (int i = 0; i < data.length; i++) {
           hash = (hash ^ data[i]) * HASH_SIZE;
       }
       hash += hash << 13;
       hash ^= hash >> 7;
       hash += hash << 3;
       hash ^= hash >> 17;
       hash += hash << 5;
       return hash;
   }

   public Hash() {}
}
```

Figure 8: (left) Ground-truth Java class and (right) generation from our CodeT5 model. Our model uses a hexadecimal value of 0x5bd1e995 (1540483477 in decimal) instead of 166136261.

```java
import java.util.Collection;
import java.util.LinkedHashMap;
public class ResultTable {
   long[] t;

   LinkedHashMap<Object, String[]> metric_values = new
LinkedHashMap<Object, String[]>();
   int length;

   public ResultTable(int length, final Collection<Object>
obj) {
       this.length = length;
       t = new long[length];
       for (Object o : obj) {
           metric_values.put(o, new String[length]);
       }
   }
   public Collection<Object> getObjects() {
       return metric_values.keySet();
   }

   public void setTimeAt(int pos, long time) {
       t[pos] = time;
   }

   public long getTimeAt(int pos) {
       return t[pos];
   }
   public void setValueForAt(Object o, int pos, String
value) {
       if (metric_values.get(o) == null) {
           throw new IllegalArgumentException(o + " is not
available in the table");
       }
       metric_values.get(o)[pos] = value;
   }

   public String getValueForAt(Object o, int pos) {
       return metric_values.get(o)[pos];
   }
}
```

```java
import java.util.Collection;
import java.util.LinkedHashMap;
public class ResultTable {
   long[] t;
   LinkedHashMap<Object, String[]> metric_values;
   int length;

   public ResultTable(int length, Collection<Object>
objects) {
       this.length = length;
       t = new long[length];
       for (Object o : objects) {
           metric_values.put(o, new String[length]);
       }
   }

   public Collection<Object> getObjects() {
       return metric_values.keySet();
   }

   public void setTimeAt(int pos, long time) {
       t[pos] = time;
   }

   public long getTimeAt(int pos) {
       return t[pos];
   }

   public void setValueForAt(Object o, int pos, String
value) {
       if (metric_values.get(o) == null)
           throw new IllegalArgumentException(o + " is not
available in the table");
       metric_values.get(o)[pos] = value;
   }

   public String getValueForAt(Object o, int pos) {
       return metric_values.get(o)[pos];
   }
}
```

Figure 9: (left) Ground-truth Java class and (right) generation from our CodeT5 model. Our model does not initialize the LinkedHashMap as it is declared.