We thank all the reviewers (R4,R5,R6) for the valuable feedback.

Difference between semantics and functionality (R6 & R5). Functionality (or correctness) of the code describes whether the behavior of the recovered code matches the low-level code. Semantics (or readability) is defined as the similarity between the recovered code and original high-level (HL) code. Our experiments (Appendix F, line 324-331) show that existing decompilers hardly satisfy neither functionality nor semantics. Their recovered programs are hard to interpret and dissimilar to the source programs. Assembly code may be incorrectly translated into the assembly operations in HL code. For example, `mov` instructions are likely to be translated into variable copy which is apparently redundant. Note that the semantics of the generated HL program is insensitive to recovered token while the functionality can be heavily affected (R6).

Metric selection/Code examples/Table explanation (R6). Token accuracy is similar to the BLEU score for NLP evaluation. It shows how close the decompiled program and the original HL program are. Program accuracy measures the percentage of recovered programs that preserve both functionality and semantics. These metrics are also used in other program translation or synthesize works ([14, 36]). We provide code examples in supplementary materials (Appendix E, F). Table 1 shows the **token accuracy** after Stage 1. The final **program accuracy** after Stage 2 is reported in Table 2. As such, the results in these two tables shall not match.

Key Motivations (R6). Our key motivations: i) Our *end-to-end* design makes the decompilation task more efficient and extensible. With the growing amount of new programming languages (PLs), new PL features (e.g. software obfuscation) and various hardware (TPU/GPU/FPGA/Accelerators), current decompilers are very limited in their usage and incur high engineering overhead. ii) Coda maintains both the functionality and semantics of the original HL program.

Solutions to syntax errors (R6). Coda trains the auto-encoder to let the neural network learn the grammar of the HL language (line 41) without linguistics knowledge. Our result (Table 1) does not show the significant disadvantage of containing a lot of syntax errors. We encounter only a few variable usage errors that lead to decompilation failure because Coda automatically learns the number of variables and their types. For AST decoding method, the generated AST is guaranteed to be compilable. A small portion of syntax errors exists in the sequence decoding baseline and we use a script to check and fix these syntax bugs. The numbers in Table 1 are measured before the script checking. Also, the error correction stage guarantees that the sketch code from stage 1 is fault-tolerant. Other errors that do not influence the compilation can be corrected in the second stage.

Recover complicated structures (R6). We use similar benchmarks as the previous decompiler works [Phoniex USENIX'13], including function calls, normal expressions, nested control graphs, variables with different types and data dependencies. Note that existing decompilers also fail to recover complicated data structures/classes. The type and structure identification is an individual research direction which has been widely studied in previous works [3][REWARD NDSS'10]. Coda is sufficient to resolve real-world applications such as Pytorch API or Hacker's delight applications. Combining Coda with these works can recover more complicate programs.

Unclear methods (R5): 1) AST tree decoder. The states (h,c) from a given AST node will feed into the left/right LSTM to generate the left/right child, as shown in Eq. (2) and (3) (line 201-210). These two expanded nodes will become the new parent nodes to generate its children using the left/right LSTM. The obtained binary AST tree (left-child right-sibling representation) will be transferred back to its equivalent AST tree. **2) Error Predictor Training.** The training target is the error type of the given AST node that we manually injected. If the error type for the given node is a mispredicted error, the EP also outputs the correct substitution token (line 232). The training loss between the EP's outputs and the targets is minimized. The training data is fixed during the EP's training.

Experiment with numerical representation (R4): The numerical representation is an existing issue in NLP applications and solutions have been proposed in other works. In our case, most of the numeric is the offset addresses that appear frequently. We treat them the same way as other tokens. The numeric can also be represented as a real-value scalar. Our experiments show for small memory footprint programs (number of variables = 10), the numerical representation in different encoding format does not lead to significantly different performance (Scalar encoding: +0.9% program accuracy on average). With an increasing number of variables and memory usage (variables = 20), scalar format shows more scalability (+2.7% program accuracy).

Experiment with longer codes (R6): With average code length (L) of 45/60, the token accuracy drops by (seq2seq: -5.4%/-13.5%,inst2ast: -3.1%/-8.4%) on average compared to $L = 30$ across benchmarks. For longer codes, instruction encoding shows better performance compared to seq2seq model. The challenges to decompile long programs are: i) Unlike natural language with period as the end of sentence, there is no clear boundary to divide assembly code. The length of the input tokenized assembly grows to a very large value (Appendix A.2). One possible solution to this problem is to divide the code using function entry point. ii) the GPU memory is not enough to train the network with a large batch size for tasks with extremely long encoding sequences.