Addressing R1, R2, and R3: We thank all the reviewers for their valuable feedback. We will incorporate the suggestions and provide a more comprehensive discussion of related work in the final version of the paper.

The reviewers raised concerns that the line-by-line alignment between pseudocode and code in our setting is too restrictive. We want to first point out that our framework is much more general: we only assume that the desired program can be decomposed into blocks, each aligned to a segment of natural language. Unlike treating the whole program as one big chunk (too coarse) or decomposing into individual tokens (too fine-grained), semantically coherent code blocks form a natural abstraction that can still be described precisely and succinctly by natural language. To instantiate this framework, we created the SPoC dataset, in which most blocks are single lines for simplicity. However, some blocks contain multiple lines, in which case the description tends to become more higher-level. For example:

(a) read n values into array a and array b $\rightarrow$ `for (int i = 0; i < n; i++) cin >> a[i] >> b[i];`
(b) print all elements of ans $\rightarrow$ `for (int i = 0; i < ans.size(); i++) cout << ans[i];`

Moreover, even the translation of each statement is non-trivial. Many blocks in the dataset are of comparable complexity to many code generation and semantic parsing tasks, and several statements are context-dependent (e.g., (b) above requires knowing the type of `ans`). With these challenges, the dataset is already difficult, and we believe this is a good initial step to tackle the problem of combining natural language and test cases in a more complex setting than any previous work. Over time, we agree it would be good to increase the difficulty of datasets by increasing the block size and the vagueness of natural language. Finally, we believe even the current system could be useful in education (getting hold of syntax), programmer productivity (moving to a new language), accessibility (coding via speech), etc.

Addressing R1's comments: R1 noted that using error detection to improve generation is not a new idea. We are aware of incremental and execution-guided decoding, which are quite established in the semantic parsing literature, and we will add appropriate citations accordingly. Nonetheless, one distinction of our prefix pruning is the focus on minimizing the number of program compilations by selectively compiling prefixes that are likely to be erroneous. In our initial attempts where we compile every prefix in the same fashion as Wang18, the generation process slows down significantly. This prompted us to look beyond whether the prefix successfully compiles and use novel signals (e.g., compilation error message) to synthesize more programs with the given budget.

In the multiclass error detection model, while positional embedding allows us to softly rule out lines that are less likely as R1 suggested, our main rationale for positional embedding is that many types of errors typically occur at a specific offset from the offending line (e.g., "'else' without a previous 'if'" mostly occurs 2 lines after the actual offending line).

Transferring to unseen problems is arguably more difficult because the model has to potentially generalize to C++ structures or methods that are specific to those problems. We will add more analysis in the next version of the paper.

R1 argued that the ability to evaluate program decoding with strong supervision is not a crucial research idea. We first want to clarify that the test cases are not only used for supervision at training time; they are also fed as input to the system at test time to (1) allow the system to search for a correct program, and (2) evaluate the functional correctness of the generated program. For (1): having test cases at test time makes decoding a more challenging task than previous semantic parsing and language-to-code settings, as the system has to perform search over a combinatorially large space of code—the fundamental challenge of program synthesis. For (2): we want to emphasize the importance of functional correctness. Besides functional correctness being ultimately necessary for generating *working* programs, partial metrics like BLEU can be gamed by learning the code boilerplate, or by mapping parts without properly combining them.

Methods that use syntax and semantics to guide synthesis for the *whole* program (NB: most of our line translations are already syntactically correct) are not guaranteed to generate semantically correct programs or even compilable programs, as only some semantic information (e.g, variable types) can be tracked during decoding. To generate semantically correct programs, we still have to search over the space of possible programs, which can be complicated or inefficient under syntactic/semantic decoding constraints when the program is large, but is an interesting direction to explore.

Addressing R2's comments: Figure 1 is indeed representative of the granularity of pseudocode. We found that a coarser granularity leads to underspecified natural language descriptions (e.g., "run Euclid's algorithm on m and n").

The ratio of token counts between pseudocode and code is roughly 1:1.2 which is lower than NAPS synthetic data.

Regarding annotation: we asked crowdworkers to give one annotation for every block of code (single C++ statements and common compound statements) in the program, which ensures that the granularity of pseudocode annotations are fixed to be identical to how we separate the code into blocks. Additionally, the whole program is visible to the crowd workers and they are indeed allowed to take the surrounding context into consideration during the annotation process.

Since our emphasis is on the search techniques, we translate each line separately for simplicity, but a context-dependent translation system could be easily plugged in. We rely on search to eliminate candidates that do not fit in context.