CODE MEANS MORE THAN PLAIN LANGUAGE: BRINGING SYNTAX STRUCTURE AWARENESS TO ALGORITHMIC PROBLEM SOLUTION GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Program Synthesis (PS) is the task of building computer programs that satisfy problem specifications. Large-scale pre-trained language models treat the PS as a sequence prediction task, which has gained vivid popularity recently. However, these methods heavily rely on the conventional Natural Language Processing (NLP) tokenizers, which *overlooks the rich structural/syntax information in the code*. In this work, we posit that the syntax structures help generate syntax error-free and algorithmically correct programs. If the program syntax structures can be integrated into the tokenizer, the program representation space could be significantly simplified. To this end, we propose a new end-to-end framework named **Syntax Aware Transformer**, coupled with our novel **syntax-aware tokenization design toolkit**. More specifically, our tokenizer encodes and decodes the program by its syntax roles and contents, not by what is superficially shown on the strings. The Syntax Aware Transformer encompasses a novel *sample-wise and token-wise attention mechanism*, and avails the benefits of training with the syntactically aligned samples from our tokenization toolkit. Extensive evaluations show superior performance against state-of-the-arts, which confirms that bringing syntax knowledge into the language model can help better capture the data structure and simplify the search space. All of our codes will be publicly available upon acceptance.

1 INTRODUCTION

The *program* has long dominated the modern industry. The modern scale language models have demonstrated the prospectives to automatically analyze, annotate, translate, or synthesis a program (Austin et al., 2021; Hendrycks et al., 2021; Chen et al., 2021; Clement et al., 2020; Wang et al., 2021; Chen et al., 2018b; Bunel et al., 2018; Devlin et al., 2017; Gulwani et al., 2012; Fox et al., 2018; Ganin et al., 2018). Among these learning tasks, the Program Synthesis (PS), also widely called as programming by example (PBE), can be defined as the task of developing an algorithm that meets a specification or a set of constraints.

The classical approaches to program synthesis date back to *rule-based program synthesis* which use formal grammar to derive programs from well-defined specifications (Waldinger & Lee, 1969; Manna & Waldinger, 1971; 1980). More recently, symbolic and neuro-symbolic techniques (Balog et al., 2017; Odena & Sutton, 2019; Ellis et al., 2018; 2020; Devlin et al., 2017; Panchekha et al., 2015) have been explored, but they have been widely applied to restricted domain-specific languages (DSLs), which limits algorithm capability. Ulike DSLs, modern general-purpose languages such as Python require handling high-level control flows (eg. loops and branching) and low-level operations, and have enormous search space where just a small perturbation of the pro-

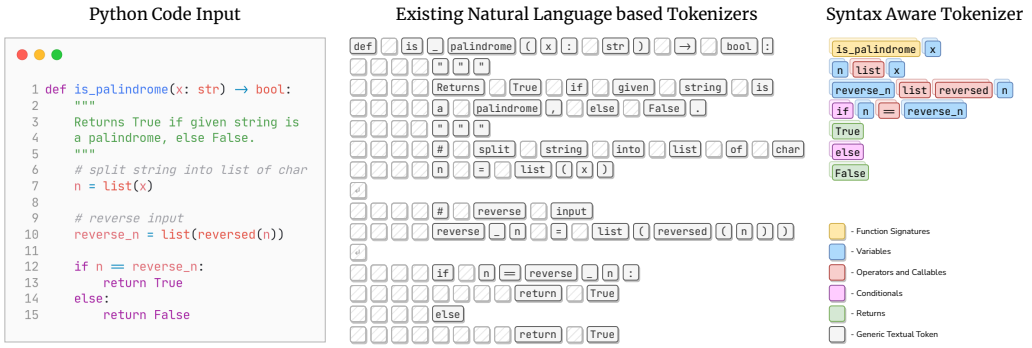


Figure 1: Tokenizers' behavior illustration: given a python code snippet (left), a natural language-based tokenizer (middle) outputs different token patterns than the proposed syntax aware tokenizer (right). The natural language tokenizer will encode a few patterns not related to algorithmic performance, such as comments and doc-strings. The spaces and variable names are also encoded with redundancy, e.g., one python indent could be encoded into four space tokens, and the variable names could sometimes be broken into multiple sub-words. The syntax-aware tokenizer automatically removes the comments and doc-strings, and recognizes the variable names according to their syntax roles, as well as a few other features. Two layers on the right means syntax subtokens and content subtokens, respectively.

gram often leads to a complete change of the output. To maneuver through this enormous search space induced by high-level languages like Python, another series of work frames program synthesis as a sequence prediction problem (Kanade et al., 2020; Feng et al., 2020; Clement et al., 2020; Svyatkovskiy et al., 2020; Wang et al., 2021) leveraging large-scale models such as transformers. Despite their great successes, these models usually leverage general purpose tokenizers, and heavily rely on conventional natural language processing (NLP) pretraining techniques on source code by merely regarding them as a sequence of tokens. These tokenizers are built on plain texts and are **overlooking the rich structural/syntax information** in code, which are very crucial for generating well-structured, syntactically correct, and executable programs.

This paper argues an under-explored theme: *can we elegantly embed syntax structural knowledge into the program synthesis paradigm with the hope of reducing the search space, regularizing representation structure, and facilitating learning?* Our question originates from one straightforward observation: programming languages have rigid syntax requirements in order to pass the compilers/interpreters and execute; while on the other hand, natural languages do not have such harsh requirements. However, previous language models indistinguishably tokenize the program data as structureless plain texts, using natural language tokenizers. We hence posit that, if syntactical knowledge could be more tightly baked into the learning procedure, the resultant model has a potential to off-load from understanding and predicting **both** *syntax* and *content*, to **focusing** solely on the *content*.

To this end, we propose **ASTer**, an integrated *syntax aware tokenization design toolkit*, together with a novel transformer model that fits on the syntax knowledge encoded tokens. As illustrated in Fig. 1, with the syntax structure knowledge baked in, the encoded sequence becomes shorter, and has its complex syntax structure simplified in a structured way. This eases the task of learning and optimization of the transformer. Regarding the transformer model, in addition to the token-wise attention mechanism, we propose a *sample-wise attention mechanism* in the encoder to learn attention weights across samples of the training instances, and generate unified vector representations regardless of sample numbers. Our technical contributions can be summarized as:

- We present a tokenization toolkit for program data that deeply *integrates the syntax structure knowledge into the language modeling* for program synthesis, instead of treating the program as a structure-less string. With the proposed tokenizer, the program data is encoded into much shorter sequences, and yet with smaller vocabulary sizes, which greatly reduce search space and facilitate learning.
- We propose a new transformer architecture, which can handle varying sample numbers as well as varying sample sequence lengths simultaneously. Combined with our tokenization toolkit, the ASTer is able to be trained on syntactically aligned data.
- To fully activate the expressiveness of ASTer's syntactically aligned representation, we propose a principled data augmentation approach that handles diverse distribution of algorithmic data, and enables training with non-static data. Extensive experiments show that ASTer reaches state-of-the-art purely with program I/O and without natural language description of the task.

2 Related Works

Synthesizing programs from description and IO pairs is a widely well received benchmark (Chen et al., 2018b; Bunel et al., 2018; Devlin et al., 2017; Gulwani et al., 2012; Fox et al., 2018; Ganin et al., 2018), yet is challenging due to an indefinite search space. Recently there has been a huge surge in exploiting neural networks for program synthesis with extension to general-purpose programming languages like Python. For example, Devlin et al. (2017) uses an encoder-decoder style neural network formulating synthesis process as a sequence generation problem, to significantly outperform non-neural program synthesis approaches on FlashFill task. With the advent of large-scale pre-trained language models (LMs), (Austin et al., 2021; Hendrycks et al., 2021; Chen et al., 2021; Clement et al., 2020; Wang et al., 2021) use transformers (Vaswani et al., 2017) to receive input sequence as problem specification in natural language and generate the sequence of code as output capitalizing the contextual representations learned from massive data of codes and natural languages.

Examples include the CodeT5 (Wang et al., 2021), which is built on T5 architecture, leveraging the code semantics conveyed from the developer-assigned identifiers and incorporates the special characteristics of programming languages such as token types. It prominently focuses around understanding tasks such as code defect detection, translation, and clone detection. OpenAI Codex Chen et al. (2021) uses GPT-3 architecture, evaluating its synthesis performance on a new benchmark of simple programming problems. CodeBERT Feng et al. (2020) which is a bimodal pre-trained for natural language and programming language like Python, Java, JavaScript, etc. captures the semantic connection between natural language and programming language.

More recent programming synthesis works such as APPS (Hendrycks et al., 2021) and AlphaCode (Li et al., 2022) have shown promising program generation results over interview/contest difficulty level questions. Yet they still adopt vanilla transformer models and natural language based tokenizers, which overlook the strict syntax of the programs. More related works on programming synthesis are discussed in our Appendix A.1.

3 The Syntax Aware Tokenization Algorithm

We systematically discuss the motivations, benefits, and approaches to build syntax structure knowledge into the program tokenizer. We take Python3 as the target programming language to be generated: when it comes to the syntax, we by default are discussing Python3 syntax henceforth.

3.1 REVISITING THE NATURAL LANGUAGE TOKENIZERS AND THE MOTIVATION FOR SYNTAX AWARE TOKENIZER

Language models like BERT, GPT, etc. have recently shown staggering transferability for programming language related tasks such as code synthesis, retrieval, translation, classification, completion, and program repair, and program synthesis, etc. (Li et al., 2022; Wang et al., 2021; Hendrycks et al., 2021; Chen et al., 2018a; Austin et al., 2021; Clement et al., 2020). Almost all of these benchmarks leverage the natural language tokenizers to encode and decode programs. These tokenizers treat the syntax-rich code snippets as naive strings of plain texts. Though such treatments have become standard and popular in practice nowadays, we note several key gaps between a program sequence and a plain text sequence.

Existing gaps. First, there are lots of user-defined names in the program, such as `'def calculate_the_index_of_the_peak_value()'`. The natural language tokenizer might break the function name into multiple sub-words, making the sequence longer. Or, the tokenizer might include the entire string `'calculate_the_index_of_the_peak_value'` as one token into its vocabulary, making its vocabulary size gigantic. Both treatments will lead to extra training difficulty. Similar cases also include the class names, local variable names, and function argument (function input variable) names. We note that the curse of user defined names is simply caused by human programmers, not by the PS task itself. If we change these names to follow another pre-defined pattern, it does not affect the algorithmic meaning and the execution results, and can largely simplify the representation.

Second, there are some patterns in the program data that do not contain algorithmic meanings but will affect the string length. Examples include the doc-strings, comments, non-indent spaces (`cnt=1` vs. `cnt = 1`), empty lines, etc. These patterns ubiquitously exist in the training set, and can vary sample by sample. They make the dataset noisy and disturb the language model from learning the truly important algorithmic patterns. The natural language tokenizers are not designed to handle these patterns, and will faithfully transit these noises to the language model.

Last but not least, the natural language and the programming language have drastically different tolerance in the string permutation: for natural language, slightly modifying the wording choice and order, minor spell errors and so on, will not affect the meaning of the sentence. In programming languages, these are not the case: any neighborhood sentence re-order, small spelling change, adding or dropping sentences will lead to significantly different algorithms and the execution results, or simply result in syntax error, name error, runtime error, etc. In summary, **the natural language tokenizer tends to overlook some critical aspects, while magnifying other "non-important" patterns (in the algorithmic execution sense)**.

Motivated by these aforementioned gaps, in this work, we design the program data syntax-aware tokenizer in lieu of the natural language-based ones. With syntax structure knowledge, the proposed tokenizer is able to identify and focus on the patterns truly tied to algorithmic meaning. For the non-algorithmic meaning-related patterns (user-defined variable name, doc-strings, comments, non-indent spaces, empty lines, etc.), it will degenerate these cases into the same token sequence representation. In the above long function name example, the proposed tokenizer will always tokenize it into one *single* word, e.g., the `'function_name_4'`, which is one of the variable name holder token from a pre-defined name pool.

The benefits of syntax aware tokenization. We see several potential benefits of this syntax-aware tokenization, as it reduces the program search space, regularizes the input space structure, and facilitates learning. First, thanks to the degenerated variable names, the vocabulary size could be largely reduced. For example, if there are two variables called `student_num` and `student_nums` within one program, the tokenier might tokenize them into two distinct word `user_defined_var_7` and `user_defined_var_8` drawn from the name pool instead

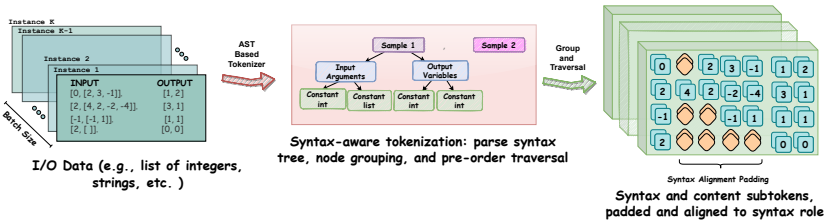


Figure 3: The syntax aware tokenization for the I/O data. Same as programs, the I/O data is also parsed into the syntax tree before tokenized. The syntax awareness also enables syntax role alignment-based padding, instead of naive tail padding or front padding.

of `[student, _, num]` and `[student, _, nums]`, which have common prefixes, and could take extra attention budgets.

On the other hand, due to the doc-string and comment removal, and the automatic detection of syntax entity without breaking any entity into multiple sub-words, the token sequence could be shortened too. In this way, the output program search space could be reduced dramatically. Besides the output program search space simplification, syntax awareness can also regularize the input space during padding. Since the input to the model is the program I/O data with strict syntax structure, when we pad across samples with different sequence length, the tokenizer is able to maximally align the syntax role, as shown in Fig. 3, unlike natural language tokenizers that pad indistinguishably to the end or front across samples. Next we discuss the details of the tokenization algorithm.

3.2 The Syntax Aware Tokenization Algorithm

To capture the syntax structure of the programming language, we resort to the abstract syntax tree (AST), a python built-in package. An example subtree is shown in Fig. 2.

```
Assign(
  targets=[Name(
    id='reverse_n',
    ctx=Store())],
  value=Call(
    func=Name(
      id='list',
      ctx=Load()),
    args=[Call(
      func=Name(
        id='reversed',
        ctx=Load()),
      args=[Name(
        id='n',
        ctx=Load())],
      keywords=[])],
    keywords=[]),
  type_comment=None)
```

Figure 2: A sub part of the AST parsed syntax tree, from the sentence `reverse_n = list(reversed(n))` in the is-palindrome example. The full tree is in Fig. 9 of Appendix (106 lines in total).

Given the parsed tree from AST, our goals are as follows: (1) design an encoding algorithm to serialize this tree representation into token sequences, which preserve full information of the original tree; and (2) design the paired decoding algorithm to convert a given token sequence back to the original code. In fact, when we view this tree as a string, it contains two parts, the *content* part, such as the `reverse_n`, `list`, `reversed`, `n`; and the syntax part which is "everything else". Therefore, we design the sequence structure as the interleaved *syntax subtokens* and *content subtokens*, with the syntax subtokens gluing the content subtokens. It is easy to observe that the relationship always satisfies: `number of syntax subtokens = number of content subtokens + 1`. A complete illustration of syntax subtokens and content subtokens are shown in Table 4, where this relationship can be more clearly visualized.

Given the syntax subtoken and content subtoken design, the decoding algorithm is obvious: first interleave the syntax and content subtokens, then glue them together, and finally unparse back to code using the AST built-in package. However, the encoding algorithm is non-trivial nor out-of-the-box due to several reasons.

First, the syntax tree is lengthy (already 106 lines for this simple is-palindrome program), mainly due to the nested syntax subtree structures. Therefore,

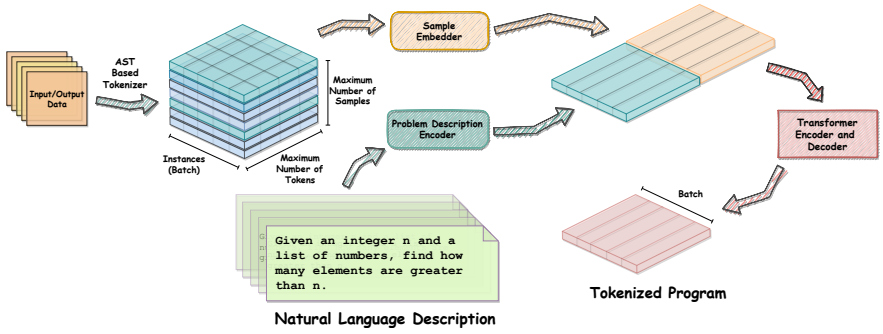


Figure 4: Overall Model Architecture of ASTer, which consists of the sample embedder, the problem description embedder, the traditional transformer encoder and decoder. The description encoder is set as optional.

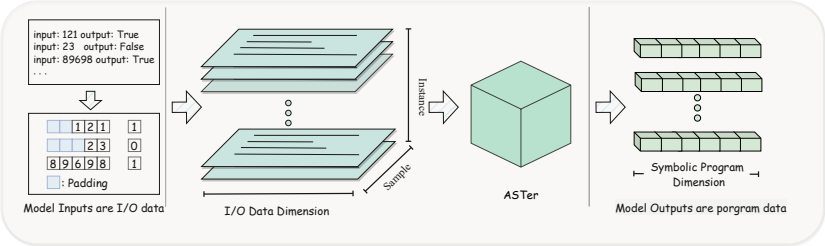


Figure 5: The input and output of the entire workflow: both the I/O data and the program are parsed, and the I/O data are further padded with syntax roles aligned.

a grouping trick must be developed to maximally compress the syntax component. Second, we have to treat this tree as a python object, not as a string; otherwise it will be difficult to separate the content subtokens and syntax subtokens (since they are all strings). What's more, when we replace the user-defined names from pool, we should avoid built-in names, imported names, and all names that are tied to it. For example, in `result = []; ...; print(result, file = dump_path)`, the name `print` and `file` should not be replaced otherwise the functionality will change / be lost, on the other hand, the names `result` and `dump_path` are defined by the programmer, hence should be changed from name pool. As another example, in `import numpy as np; x = np.random.randint(n, size=4)`, the names `numpy`, `np`, `random`, `randint`, `size` are tied to `import` sentence, hence should not be replaced, while `x`, `n` should be replaced; in comparison, in `my_generator.random.randint(n, my_size=4)`, the name `my_generator` is a previously user-defined class name, so all of `my_generator`, `random`, `randint`, `my_size` *should* be replaced. Give a tree like in Fig. 2, we cannot judge whether a name should be replaced or not simply by looking at the name itself, because all names appear as strings. Instead, we should leverage the syntax object structures to properly handle these challenges. To this end, we develop an algorithm named Group-On-First-Leaf-Child-Met (Goflec), shown in Table. 1.

Another issue when building the tokenizer is that it is difficult to predict the tokens. This is straightforward to observe from Fig. 2: the syntax subtokens follow highly non-standard distributions (check Table. 4 for a more clear illustration), and the content subtokens can include a broad variety of imported names and import-tied names. To address this, we do a sweep across all samples in the training set prior to model training, update the vocabulary with all syntax and content subtokens encountered and set the vocabulary as fixed since then.

4 TRAINING THE SYNTAX AWARE TRANSFORMER

In this work, we aim to explore and demonstrate: if the tokenizer is designed properly, even with *only* the I/O data without the natural language problem description, the model can learn to make correct predictions. To achieve this goal, the model design should also be carefully designed, and paired with the tokenization algorithm. Next we discuss the model architecture part of ASTer.

4.1 THE MODEL ARCHITECTURE

Unlike traditional NLP, the PS task require mapping a two-dimensional sequence into one dimensional sequence (the I/O data has both sample dimension and token dimension). We accomodate this with a new transformer architecture that carefully encodes such informations. The input/output is shown Figure 5, and the overall model architecture is shown in Fig.4, which consists of four components.

Sample Embedder: The sample embedder is provided with I/O data that consist N_{inst} instances. Each instance is represented as a matrix $W_n \in \mathbb{R}^{N_{\text{sample}} \times N_{\text{token}} \times E}$, where N_{token} denotes the maximum number of tokens, N_{sample} is the maximum number of samples within the N_{inst}, and E represents the token encoding dimension.

As illustrated in Figure 7, the content and syntax subtokens are added with two positional embeddings. The tokenizer pads the empty token dimension and the sample dimension to the maximum of the corresponding in each batch. This ensures the same syntax role tokens are aligned across samples. Through first element pooling, the output of the sample embedder is reduced to $N_{\text{token}} \times N_{\text{inst}} \times E$ dimension.

Description Embedder: Each instance in the I/O data corresponds to a problem description. We utilized and fine-tuned two existing pretrained language models to encode them, and output four distillation tokens: first element, last element, minimum, and maximum pooling. We make this submodule is optional since access to the problem description could be limited.

Token Encoder, Program Decoder And Loss: After concatenating the problem description and sample embedder outputs, it then follows traditional transformer encoder and decoder layers. At the output, we ask the model to predict a interleaved syntax and content subtokens.

4.2 THE DATA AUGMENTATION

To avoid the model from overfitting the training data, and to help to learn the underlying algorithmic patterns of I/O to program transitions, we employ a principled way of generating new data each time. The general principle is to carefully generate I/O data following mixed distributions, then execute the program to get the output. We discuss this in more details at Appendix A.3.

5 EXPERIMENTS

5.1 DATASETS AND TOKENIZATION RESULTS

Our training instances come from the training set of two benchmark datasets: the code contests (Li et al., 2022) and APPS (Hendrycks et al., 2021). The code-contests comprises of data scraped from Codeforces with existing data from Description2Code (Caballero et al., 2016). The APPS includes code and I/O data from Codewars, AtCoder, Kattis, and Codeforces. Our test set follows the same setting as the APPS test set, where every instance is tagged with a difficulty level of "introductory", "interview" and "competition".

We first sweep across these two datasets to collect vocabulary, yielding 10048 instances in total. Here the instance refers to one problem with one natural language task description, multiple I/O data pairs and multiple code solutions. We refer to this new dataset as STTD (Syntax Tokenization

Transferred Dataset). We use the APPS test set part for testing and the rest as our training set. The number of samples per instance statistics of the STTD are in the first two columns of Table. 1.

After the sweep, we tokenize the program data in APPS and contests, and collected the sequence length in the first two columns of Table. 1. We found that the proposed syntax-aware tokenizer encodes the program into shorter sequence, compared with the BERT tokenizer, for example, in the APPS dataset, the average sequence length is 78.404 syntax subtokens + 77.404 content subtokens, whereas BERT tokenizer on average generates 176.312 tokens. Note that the syntax subtoken number is always content subtoken number plus one (see section 3.2), and a language model could enjoy this almost half-length sequence, just by using two projection heads for the transformer decoder last hidden layer (one for syntax subtoken, the other for content subtoken). Though we still use one output projector and train ASTer with doubled sequence length, the sequence length (i.e., 156.808 for APPS) is still smaller than that of BERT tokenizer 176.312.

In addition to the sequence length, the frequency of the syntax subtokens are shown in Figure 6. Note that the y-value of the right half of tokens is one, and the majority y-value is small. This means that only a tiny portion of tokens are truly frequent, and the model will be able to predict the majority of syntactically correct programs, so long as it has learned how to use this small subset of tokens on the left side.

	STTD Num. Samples Per Instance		APPS (Hendrycks et al., 2021) Encoded Seq. Length Per Sample			Contests (Li et al., 2022) Encoded Seq. Length Per Sample		
	I/O	Program	ASTer	BERT	Raw String	ASTer	BERT	Raw String
Mean	73.683	140.065	78.404	176.312	416.419	73.338	176.509	432.566
Median	82.0	25.0	57.0	124.0	278.0	56.0	124.0	289.0
Std.	43.480	450.478	272.911	261.470	618.100	82.547	292.139	780.052
Count	10048	10048	81339	81339	81339	1337655	1337655	1337655

Table 1: The STTD dataset statistics (per-instance, left two columns) and the tokenization sequence length statistics (per-sample, right six columns). The syntax-aware tokenizer sequence length is calculated by the number of syntax subtokens.

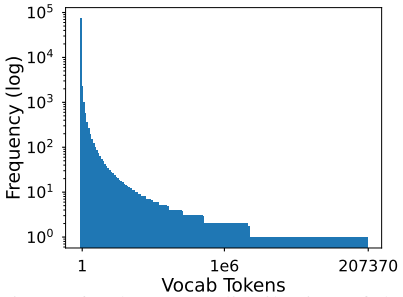


Figure 6: The count distribution of the syntax subtokens encountered (higher value means more frequent tokens).

5.2 Models and Hyperparameters

We train two versions of Syntax Aware Transformer: The ASTer$^{\mathrm{I/O+BERT}}$, which has access to both the textual description of the question as well its corresponding I/O data, and ASTer$^{\mathrm{I/O}}$, which only have access to the I/O data. For the first one, we employ the BERT (Devlin et al., 2018) to process descriptions and fixed its parameters during the first 5×10^4 training instances. The rest part of the model as well as the entire ASTer$^{\mathrm{I/O}}$ is trained from scratch. Both models have four sample embedder layers, four transformer embedder layers, eight transformer decoder layers, the hidden dimensions of sample embedder and encoder are both 1024, and the decoder hidden dimension is 768. At inference, we set the beam search width to be 5. We follow the same evaluation as in APPS and Alphacode on the APPS test set. The results are shown in Table. 2.

As seen in Table. 2, ASTer$^{\mathrm{I/O}}$ reaches the **state-of-the-art** performance, with much fewer sampling numbers from the transformer (5 beams as opposed to 50000 in Alphacode). In addition, the ASTer$^{\mathrm{I/O+BERT}}$ model performs worse than ASTer$^{\mathrm{I/O}}$. This is possibly due to overfitting the

	Filtered From(k)	Attempts (n)	Introductory		Interview		Competition	
			n@k	syntax pass	n@k	syntax pass	n@k	syntax pass
GPT-3 few shot	N/A	1	0.20%	31.0%	0.03%	42.0%	0.00%	40.0%
GPT-Neo 2.7B	N/A	1	3.90%	87.9%	0.57%	87.9%	0.00%	85.0%
GPT-Neo 2.7B	N/A	5	5.50%	97.4%	0.80%	96.0%	0.00%	95.4%
AlphaCode 1B	50000	5	20.36%	N/A	9.66%	N/A	7.75%	N/A
ASTer$^{\mathrm{I/O+BERT}}$ (16 samples)	5	1	6.5%	**100%**	0.75%	98.5%	0.00%	96.7%
ASTer$^{\mathrm{I/O}}$ (16 samples)	5	1	**26.8%**	**100%**	**12.58%**	**98.8%**	**8.30%**	**97.5%**
ASTer$^{\mathrm{I/O}}$ (4 samples)	5	1	17.6%	**100%**	8.80%	98.6%	0.50%	97.0%

Table 2: The results of APPS Hendrycks et al. (2021), AlphaCode Li et al. (2022) and the proposed syntax tokenizer model on the APPS test set.

training data and the decoder over-rely on the natural language part, instead of the I/O data part, to make its predictions. This is because our training set is relatively small, and when we do the I/O data augmentation, the language description of the problem remains fixed while the I/O data keeps mutating. On the other hand, it also demonstrates that simply using the I/O data suffices for ASTer to infer the algorithm logic behind it.

With regard to the syntax pass (syntax error free) rate, both ASTer$^{\mathrm{I/O+BERT}}$ and ASTer$^{\mathrm{I/O}}$ stay competitive, and this number almost did not drop when the samples provided for inference is dropped from 16 to 4, while only the accuracy significantly drops. It seems that for the ASTer, *making the predictions syntactically correct* has been decoupled from *making the answer algorithmatically correct*, and whenever the input samples is not enough to infer an algorithmically correct program, the ASTer's decoder layers tends to blindly predict a random program, with correct syntax. We hypothesize this is due to the tokenizer has **greatly simplified the search space**, making the syntax error-free an easier job for the decoder.

6 Conclusion, Limitations and Discussions

This work proposed ASTer: a novel syntax-aware tokenization toolkit, together with a transformer architecture that suits the syntax-based tokens. The tokenization algorithm encodes the program using syntax tree parsing, followed by a leaf node grouping technique. The resulting syntax based tokens filter out and degenerate every information that is not related to algorithmic meaning. Experimental results have shown that ASTer has achieved state-of-the-art results with smaller model sizes and fewer sampling numbers, thanks to the degenerated algorithmic search space.

This work offers a brand new possibility to encode the program data, which analyze and parse not by partitioning what is superficially shown in the code strings. We have focused on the tokenization toolkit implementation for the python language, but we see the algorithm is general to all programming languages, and may immediately boost previous language models (Kanade et al., 2020; Feng et al., 2020; Clement et al., 2020; Svyatkovskiy et al., 2020; Wang et al., 2021).

Despite the fact that ASTer has achieved state-of-the-art performance using much less sampling numbers compared with Alphacode, and has a smaller model size, we notice that the performance come with the cost of sacrificing readability. This is because all the comments and doc-strings are removed by the tokenizer, and hence the model is unable to predict them. On the other hand, the variable names are now simply name holders (visualized in Fig. 8), unlike other models that could predict `i`, `j`, `n`, `cnt` to represent an integer, `nums` to represent an one dimensional array of numbers, etc. We see two ways to address naming blindness and poor readability deficiency. One way is to disable the name replacement step in the ASTer tokenizer, and train the model with the actual and diverse name tokens. This will bring back the algorithmic irrelevance, and make the vocabulary larger, which could possibly downgrade the model performance. Also, this way does not add back the comments and doc-strings, if they are desired. The second mitigation is to concatenate ASTer with another transformer encoder-only model. This model specializes to predict masked variable names and doc-strings given the algorithm skeleton, which is given by the ASTer output. We leave this as our future work.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net, 2017.

Rastislav Bodík and Armando Solar-Lezama. Program synthesis by sketching. 2008.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Ethan Caballero, . OpenAI, and Ilya Sutskever. Description2Code Dataset, 8 2016. URL `https://github.com/ethancaballero/description2code`.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018a.

Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Towards synthesizing complex programs from input-output examples. *arXiv: Learning*, 2018b.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis. *Advances in Neural Information Processing Systems*, 2021.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: Multi-mode translation of natural language and python code with transformers. *ArXiv*, abs/2010.03150, 2020.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS*, 2018.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *ArXiv*, abs/2006.08381, 2020.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. Parametrized hierarchical procedures for neural programming. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJl63fZRb`.

Yaroslav Ganin, Tejas D. Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *ArXiv*, abs/1804.01118, 2018.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL '11*, 2011.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. volume 55, pp. 97–105, January 2012. Invited to CACM Research Highlights.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *ICML*, 2020.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14:151–165, 1971.

Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. In *TOPL*, 1980.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2019.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Communications of the ACM*, 59:114–122, 01 2016. doi: 10.1145/2863701.

Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *ESEC/FSE '11*, 2011.

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05*, 2005.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *IJCAI*, 1969.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859, 2021.

A APPENDIX

We present a few supplementary results in the following sections. They are aimed to provide more insights into our related works, tokenization and model.

A.1 CONTINUED RELATED WORKS ON DOMAIN SPECIFIC LANGUAGE AND PROGRAM SYNTHESIS

Program synthesis tasks can be traced long back to advent of early machine learning research, where (Waldinger & Lee, 1969) used theorem prover to syntesize LISP programs based on the formal specification of input-output relations. Recently, it has again received enormous attention on the back of development of methods for learning programs spanning across multiple domains such as synthesizing data structure manipulations (Singh & Solar-Lezama, 2011), string programming (Gulwani, 2011; Gulwani et al., 2012), low-level bit manipulation (Solar-Lezama et al., 2005), etc. Predominantly, the recent approaches for program synthesis operate in a carefully engineered Domain-Specific Language (DSLs) rather of unrestrained Turing-complete languages due to enlarge the search space and complicated synthesis (Polozov & Gulwani, 2015; Balog et al., 2017; Odena & Sutton, 2019; Ellis et al., 2018; 2020; Devlin et al., 2017; Panchekha et al., 2015). Unlike full-featured programming languages (eg. Python, Java, C++. etc.), DSLs are programming languages which constrain the search over programs with strong prior knowledge in the form of a restricted set of programming primitives tuned to the needs of the domain. For instance, one might invalidate the usage of control flows or loops (eg. `if-else`,`while`, etc.), and restrict only limited primitive operations like concatenation.

Numerous searching approaches such as constraint-based (Bodík & Solar-Lezama, 2008), enumerative, and stochastic algorithms has been proposed for DSLs supporting different specifications and domains. A successful example of using stochastic local search to find assembly programs having same sementics as an input program is STOKE super-optimization (Schkufza et al., 2016). Despite some noticeable success, these techniques require ample research efforts and engineering to come up with diligently-designed heuristics for efficient search and they also suffer from limited applicability and can only generate programs of small size and restricted types (Parisotto et al., 2016; Balog et al., 2017; Bunel et al., 2018).

A.2 FURTHER DETAILS OF THE TOKENIZATION ALGORITHM

The proposed syntax based tokenizer encdoder algorithm, the Group-On-First-Leaf-Child-Met (Goflec), is displayed in Algorithm 1. This algorithm is able to maximally compress token sequence length, detect all imported names and names that tied to imported names, and remove doc-strings (in comparison, the comments are already removed by the AST package).

A.3 INPUT DATA GENERATION APPROACH

The algorithmic solutions space, given an instance with a set of I/O data samples is combinatorially large – there exist numerous solutions to the same problem. If the samples are not well distributed, or in other words do not depict a closed-form representation of the underlying causal logic, the **possibility of over-fitting is tremendously high**. A model synthesising code solely based on I/O data is *severely* limited by the quality of that data.

Algorithm 1 The Group-On-First-Leaf-Child-Met (Goflec) Algorithm

Require: Input abstract syntax tree
Ensure: Output syntax subtokens and content subtokens sequence

```
Initialization;
while Pre-order traversal not terminated do
    Get next node;
    if New node is leaf node then
        Pack all syntax subtokens seen sofar into one syntax subtoken
        Pair the syntax subtoken with content subtoken
        Output to final tokenized sequence and end while loop;
    else if Loop to the end of a complete sentence then
        Do grammar analyze to this sentence
        if This sentence is a doc-string then
            Remove the current sentence
        end if
        for All the variable names in this sentence do
            if Variable name is linked to any imported name then
                Do not replace the name;
            else
                Replace it from name pool;
            end if
        end for
    end if
end while
```

To help alleviate this issue, we carefully engineer an augmentation pipeline for the generation of more I/O data samples to reduce variance. Our overall pipeline consists of ❶ structured guessing of input data candidates, ❷ generating their corresponding outputs by passing inputs to the respective ground truth code, and ❸ filtering out I/O data samples which do not contribute towards uniformly spanning the output space. The last step is henceforth termed – distribution control. We explore each of these steps briefly.

1. **Guessing of input data candidates:** As a *true* closed form interpretation hard to achieve (as the length of such a set would be arbitrarily large spanning the complete input data space), we guess numerous candidate ones which can best represent it. This task is grows challenging very quickly. Firstly, the format of input data is different for every single problem in the dataset. There are variations in datatypes (list, int, float, str, bool), or can be arranged in very specific multi-dimensional nested formats of assorted lengths. Secondly, the range of values each of these elements can take, and the distribution they seem to follow is also to be strategically inferred – we also observe correlated distri-

Type	Definition	Example
Build in python vocabs	About 600 common python's built-in functions and keywords	"__name__", "__package__", ...
Digit	Int representation of single digit number	0,1,2,3,4,5,6,7,8,9
ASCII	Character encoding	'a', '!', '=', '0', '[SPACE]',...
Common float	Float that appears most common while training	0.1, 0.0001, 0.5, 0.574, ...
Wild token	Other tokens collected while training	'numpy', 12.231, 90000.0, ...

Table 3: Types of content token

butions which do not fit under i.i.d. Gaussian nor i.i.d. uniform. Thirdly, few samples have inputs which are dependent on one another. For instance there exist "master" inputs which control the shape of the succeeding inputs. Lastly, all such patterns are to be deduced just a few given inputs. We have carefully developed a systematic and rigorous input data generation approach capable of recognizing all such patterns in a highly controlled manner. We discuss this in detail in our Appendix.

2. **Generating corresponding output data:** With our freshly generated input samples, we now proceed to generating their corresponding outputs. This is simply done by running the ground truth solution against these inputs and saving resultant outputs. As we generate a large amount of input data points, to achieve speed-up we parallelize this step across multiple CPUs through multiprocessing.
3. **Distribution control:** Distribution control is essentially present to discard newly generated I/O data samples which add little to no value to the existing representation being built up by older I/O data.

These steps are repeated till the desired quota of I/O data is reached. At the end of I/O data augmentation we have a vastly superior set of samples in terms of their collective presence.

A.4 Sample Embedder

The sample embedder takes in the tokenized I/O data and transforms it into an embedding using a transformer encoder. Each subtoken and and their positional embeddings are added to form the sample tokens. The sample tokens are encoded using the proposed sample embedder and the ouputs are element-wise pooled to create the final embedded representation. Figure 7 provides an illustrative representation of the same.

A.5 Model Outputs

We showcase in Figure 8 the example outputs of Syntax Aware Transformer and the superiority enjoyed in terms of syntax and content over earlier work.

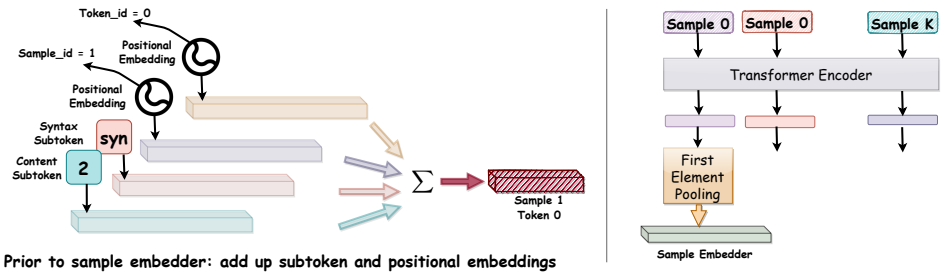
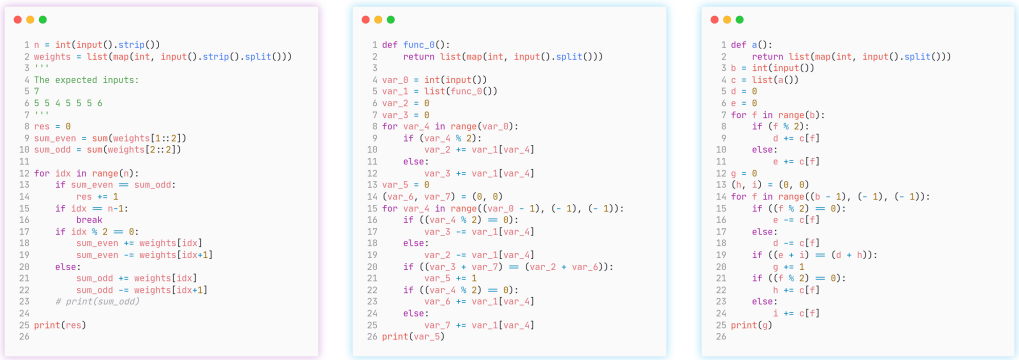


Figure 7: The architecture of sample embedder.

Sample	Input	Content	Syntax
sample 1	Input 1	❑	◊
		❑	◊
		0	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		2	kind=None),Constant(value=
		3	kind=None),Constant(value=
	Input 2	❑	◊
		❑	◊
		0	kind=None)],ctx=Load()),Constant(value=
		∅	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])
sample 2	Input 1	0	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		2	kind=None),Constant(value=
		3	kind=None),Constant(value=
		5	kind=None),Constant(value=
		1	kind=None),Constant(value=
	Input 2	❑	◊
		❑	◊
		2	kind=None)],ctx=Load()),Constant(value=
		∅	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])
sample 3	Input 1	❑	◊
		4	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		5	kind=None),Constant(value=
		6	kind=None),Constant(value=
		7	kind=None),Constant(value=
	Input 2	5	kind=None)],ctx=Load()),Constant(value=
		3	⊕
		2	⊕
		∅	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])’

Table 4: One instance with three exsamples of I/O data. `io1 = [[[0,2,3],0], [12, 'abcd']], io2 = [[[0,2,3,5,1], 2], [43, 'm']], io3 = [[[4,5,6,7], 532], [9908, 'ss']]`. The ∅ and ⊕ are inter-sample padding.



```
n = int(input().strip())
weights = list(map(int, input().strip().split()))
'''
The expected inputs:
7
5 5 4 5 5 5 6
'''
res = 0
sum_even = sum(weights[1::2])
sum_odd = sum(weights[2::2])

for idx in range(n):
    if sum_even == sum_odd:
        res += 1
    if idx == n-1:
        break
    if idx % 2 == 0:
        sum_even += weights[idx]
        sum_even -= weights[idx+1]
    else:
        sum_odd += weights[idx]
        sum_odd -= weights[idx+1]
    # print(sum_odd)

print(res)
```

```
def func_0():
    return list(map(int, input().split()))

var_0 = int(input())
var_1 = list(func_0())
var_2 = 0
var_3 = 0
for var_4 in range(var_0):
    if (var_4 % 2):
        var_2 += var_1[var_4]
    else:
        var_3 += var_1[var_4]
var_5 = 0
(var_6, var_7) = (0, 0)
for var_4 in range((var_0 - 1), (- 1), (- 1)):
    if ((var_4 % 2) == 0):
        var_3 -= var_1[var_4]
    else:
        var_2 -= var_1[var_4]
    if ((var_3 + var_7) == (var_2 + var_6)):
        var_5 += 1
    if ((var_4 % 2) == 0):
        var_6 += var_1[var_4]
    else:
        var_7 += var_1[var_4]
print(var_5)
```

```
def a():
    return list(map(int, input().split()))
b = int(input())
c = list(a())
d = 0
e = 0
for f in range(b):
    if (f % 2):
        d += c[f]
    else:
        e += c[f]
g = 0
(h, i) = (0, 0)
for f in range((b - 1), (- 1), (- 1)):
    if ((f % 2) == 0):
        e -= c[f]
    else:
        d -= c[f]
    if ((e + i) == (d + h)):
        g += 1
    if ((f % 2) == 0):
        h += c[f]
    else:
        i += c[f]
print(g)
```

Figure 8: The example outputs of APPS model (GPT-Neo 2.7B, left) and ASTer (middle and right, with two different post-naming rules), over the same problem.

```
body=[
  FunctionDef(
    name='isPalindrome',
    args=arguments(
      posonlyargs=[],
      args=[arg(
        arg='x',
        annotation=Name(
          id='int',
          ctx=Load()),
        type_comment=None)],
      vararg=None,
      kwonlyargs=[],
      kw_defaults=[],
      kwarg=None,
      defaults=[]),
    body=[
      Assign(
        targets=[Name(
          id='n',
          ctx=Store())],
        value=Call(
          func=Name(
            id='list',
            ctx=Load()),
          args=[Call(
            func=Name(
              id='str',
              ctx=Load()),
            args=[Name(
              id='x',
              ctx=Load())],
            keywords=[])],
          keywords=[]),
        type_comment=None),
      Assign(
        targets=[Name(
          id='reverse_n',
          ctx=Store())],
        value=Call(
          func=Name(
            id='list',
            ctx=Load()),
          args=[Call(
            func=Name(
              id='reversed',
              ctx=Load()),
            args=[Name(
              id='n',
              ctx=Load())],
            keywords=[])],
          keywords=[]),
        type_comment=None),
      If(
        test=Compare(
          left=Name(
            id='n',
            ctx=Load()),
          ops=[Eq()],
          comparators=[Name(
            id='reverse_n',
            ctx=Load())]),
        body=[Return(value=Constant(
          value=True,
          kind=None))],
        orelse=[Return(value=Constant(
          value=False,
          kind=None))])],
    decorator_list=[],
    returns=Name(
      id='bool',
      ctx=Load()),
    type_comment=None),
  Assign(
    targets=[Name(
      id='reverse_n',
      ctx=Store())],
    value=Subscript(
      value=Constant(
        value='121',
        kind=None),
      slice=Slice(
        lower=None,
        upper=None,
        step=UnaryOp(
          op=USub(),
          operand=Constant(
            value=1,
            kind=None))),
      ctx=Load()),
    type_comment=None),
  Expr(value=Call(
    func=Name(
      id='print',
      ctx=Load()),
    args=[Call(
      func=Name(
        id='isPalindrome',
        ctx=Load()),
      args=[Name(
        id='reverse_n',
        ctx=Load())],
      keywords=[])],
    keywords=[]))],
type_ignores=[])
```

Figure 9: Complete format of our AST tree, in the example of `isPalindrome` case.