Neural Program Generation Modulo Static Analysis

Anonymous Author(s)
Affiliation
Address
email

Abstract

State-of-the-art neural models of source code tend to be evaluated on the generation of individual expressions and lines of code, and commonly fail on long-horizon tasks such as the generation of entire method bodies. We propose to address this deficiency using weak supervision from a static program analyzer. Our neurosymbolic method allows a deep generative model to symbolically compute, using calls to a static analysis tool, long-distance semantic relationships in the code that it has already generated. During training, the model observes these relationships and learns to generate programs conditioned on them. We apply our approach to the problem of generating entire Java methods given the remainder of the class that contains the method. Our experiments show that the approach substantially outperforms a state-of-the-art transformer and a model that explicitly tries to learn program semantics on this task, both in terms of producing programs free of basic semantic errors and in terms of syntactically matching the ground truth.

1 Introduction

Neural models of source code have received much attention in the recent past [31, 8, 21, 18, 24, 29, 14, 19]. Such models have been applied to a range of programming tasks [19], including code completion, repair, translation, and search.

However, these models have a key weakness: while they can generate individual expressions or lines of code reasonably well, they fare poorly on the generation of larger blocks of code. For example, as we show later in this paper, the state-of-the-art CODEGPT model [19] frequently generates code with elementary semantic errors, such as uninitialized variables and type-incorrect expressions, when tasked with generating *method bodies* as opposed to single lines. Even in terms of syntactic accuracy measures, the quality of the generated code rapidly declines as the horizon of generation increases.

The deep-seated reason for this is that contemporary neural models of code treat programs as purely syntactic objects. In principle, the model could learn the semantics of programs from syntax, given enough data. In practice, such learning is difficult for complex, general-purpose programming languages. Prior work [7] has sought to overcome this challenge by encoding programs with bespoke graph neural network (GNN) architectures. However, as we show later, this strategy only goes so far.

In this paper, we present an alternative, *neurosymbolic* approach to the long-horizon generation of source code. Our main observation is that symbolic methods — specifically, static program analysis — can extract deep semantic relationships between far-removed parts of a program. However, these relationships are not apparent at the level of syntax, and it is difficult for neural networks to learn them automatically. Driven by this observation, we use a static analysis tool as a *weak supervisor* for a deep generative model. This means that at each point during generation, our deep model invokes the static analyzer to compute a set of semantic facts about the code generated so far. The distribution over the model's next generation actions is conditioned on these facts. We concretely develop our approach by extending *attribute grammars* [16], which are like context-free grammars but allow rules

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

to carry symbolic *attributes* constructed from the context in which a rule is fired. In our extension, called *Neurosymbolic Attribute Grammars* (NSGs), the context is an incomplete program, and rules are fired to replace a nonterminal (a stand-in for unknown code) in this program. The attributes are semantic relationships (for example, symbol tables) computed using static analysis. The neural part of the model represents a probability distribution over the rules of the grammar conditioned on the attributes. During generation, the model repeatedly samples from this distribution while simultaneously computing the attributes of the generated code.

We evaluate our approach in the task of generating Java method bodies given the rest of the code in the class in which the method occurs. To perform this evaluation, we train an NSG whose neural component is a tree LSTM and whose with grammar tracks 12 different semantic properties, on a corpus of 1 million curated Java programs over a large vocabulary of API methods and types. We compare our method against CODEGPT [19] and the method for GNN-based program encoding that we mentioned earlier [7] . Our experiments show that the NSG reliably outperforms these baselines on this task, both in terms of producing programs free of basic semantic errors and in terms of matching the ground truth syntactically.

In summary, this paper makes three contributions:

- We present a new approach to the generative modeling of source code that uses a static analysis tool as a weak supervisor.
- We embody this approach in the specific form of *neurosymbolic attribute grammars* (NSGs).
- We evaluate NSGs on the long-horizon task of generating entire Java method bodies, and show the approach to substantially outperform two state-of-the-art baselines.

2 Conditional Program Generation

We start by stating our problem, known as conditional program generation (CPG) [21]. We imagine a joint distribution $\mathcal{D}(X, Y)$, where X ranges over *specifications* of program-generation problems and Y ranges over programs. The probability $\mathcal{D}(X = \mathsf{X}, Y = \mathsf{Y})$ is high when Y is a solution to X. Also, we consider a family of distributions $\mathcal{P}_\theta(Y|X = \mathsf{X})$, parameterized by θ, that we might want to learn. *Learning to conditionally generate* programs amounts to finding parameters θ that minimize the prediction error $\mathbf{E}_{(\mathsf{X},\mathsf{Y})\sim\mathcal{D}}[\delta(\mathcal{P}_\theta(\mathsf{X}|\mathsf{Y}), \mathsf{Y})]$, where δ is a suitable distance function between programs.

Specifications and distances between programs can be defined in many ways. In our experiments, the goal is to generate Java method bodies. A specification is an *evidence set* that contains information — e.g., method names, types of variables and methods — about the class in which the method lies. We define $\delta(\mathsf{Y}_1, \mathsf{Y}_2)$ to be a large number if Y_1 or Y_2 violates one of several language-level invariants (e.g., type-safety, initialization of variables before use) that we require programs to satisfy. When both programs satisfy the invariants, $\delta(\mathsf{Y}_1, \mathsf{Y}_2)$ measures the textual dissimilarity between the two programs.

(a)

```
public class FileUtil{
  String err;
  public int read(File f){...}

  /* write lines to file */
  public void write(
    File f, String str){??}}
```

(b)

```
void write(File f, String str){
  try {
    FileWriter var_0;
    var_0 = new FileWriter(f);
    var_0.write(str);
  } catch(IOException var_0) {
    var_0.printStackTrace();
    System.out.println( ARG ); }
  return; }
```

Figure 1: (a) An instance of conditional program generation. (b) A completion of the `write` method, generated using an NSG. `ARG` is standing for a string literal.

Note that CPG is a much more challenging task than next-token-prediction, which is often studied in the literature [19, 7]. The goal is to predict long sequences of tokens (in this paper, an entire method body) and not simply the next token. Further, X is a (possibly imprecise) specification of the code to generate, not a sequence of tokens we are trying to complete by choosing the correct method to call for a variable, as in next-token prediction.

Example. Fig. 1-(a) illustrates the kind of task that we target. Here, we are given a class with a missing `write` method. The specification X here includes: (i) the class name `FileUtils`; (ii) the type `String` of the class variable `err`; (iii) information about complete methods within the class (including the methods' return types and formal-parameter types and names, and sequences of API calls made within such methods); (iv) information about the method with missing code (`write`), including its name, formal parameters, and JavaDoc comments for the method with missing code (e.g., "write lines to file"). Our objective on this input is to automatically generate a non-buggy, natural completion of `write`, without any provided, partial implementation of the method.

To understand the challenges in this task, consider a completion that starts by: (i) Declaring a local variable `var_0`; and (ii) invoking the constructor for `FileWriter` and storing the result in `var_0`. A proper implementation of these two steps must ensure that `var_0` is of type `FileWriter`. Also, the first argument to the constructor of `FileWriter` must be of type `File` (or a subtype of `File`). As we show in Sec. 5, it is hard for state-of-the-art neural models to learn to satisfy these rules.

In contrast, in our weakly supervised approach, the generator has access to a set of semantic attributes computed via static analysis. Specifically, these attributes include a *symbol table* mapping in-scope variables to their types.

Suppose that during training we are given the following line of code.

```
var_0 = new FileWriter(f, true)
```

Our model's symbol table includes the names `var_0` and `f` and also carries information about their types. The grammar is also able to compute the type of the first argument in the invoked constructor for `FileWriter`. Consequently, the model can observe that the type of `f` is listed as `File` in the symbol table, and that `f` is the first argument to the `FileWriter` constructor. With a few observations like these, the model can learn that the first argument of "**new** `FileWriter`" tends to be of type `File` (or a subtype). During generation, the model uses this knowledge, locating a variable of the correct type in the symbol table each time it constructs a `FileWriter`.

Fig. 1-(b) shows a top completion of `write` generated by our NSG implementation. Note that all variables in this code are initialized before use, and that all operations are type-safe. Also, the name `var_0` is reused between the **try** and the **catch** blocks. Such reuse is possible because the symbol table carries information about the scopes to which different names belong. Finally, as we will see in Sec. 5, the extra information provided by the static analyzer can also help with accuracy in terms of syntactic matches with the ground truth.

3 Static Analysis with Attribute Grammars

As mentioned in Sec. 1, we develop our approach as an extension of the classic attribute grammar (AG) framework [16]. Now we give some background on static analysis using AGs. In the next section, we show how to use AGs to weakly supervise a neural program generator.

An AG extends a traditional context-free grammar (CFG) [15] by attaching a set of *attributes* to each terminal or nonterminal symbol of the grammar, and by using a set of *attribute equations* to propagate attribute values through syntax trees.

The attributes of a symbol S can be divided into *inherited* attributes and *synthesized* attributes, which we suffix by $\downarrow$ and $\uparrow$, respectively. Inherited attributes transfer information from parent to child, or from a node to itself. Synthesized attributes transfer information from child to parent, from a node to a sibling, or from a node to itself. We assume that the terminal symbols of the grammar have no synthesized attributes and that the root symbol of the grammar has a special set of inherited attributes, known as the *initial attributes*.

The *output* attributes of a production $S \rightarrow S_1, \ldots, S_k$ consist of the synthesized-attribute occurrences of the nonterminal S, plus the inherited-attribute occurrences of all of the S_i-s. The *input* attributes are the inherited-attribute occurrences of S, plus the synthesized-attribute occurrences of the S_i-s. The grammar's attribute equations relate the input and output attributes of a node in terms of the attributes of its parent, children, and left sibling in the syntax tree that the grammar generates.

(a)

Stmt : *Stmt*; *Stmt*;
Stmt : *Expr* . *Method* (*ArgList*)
| *DeclType Var* = **new** *NewType* (*ArgList*)

(b)

Stmt { *SymTab* symTab ↓; *SymTab* symTabOut ↑; };

(c)

Stmt : *Stmt*; *Stmt*;
[*Stmt*\$1.symTab ↓ := *Stmt*\$0.symTab ↓
Stmt\$2.symTab ↓ := *Stmt*\$1.symTabOut ↑
Stmt\$0.symTabOut ↑ := *Stmt*\$2.symTabOut ↑]

Stmt : *Expr* . *Method* (*ArgList*)
[*Stmt*.symTabOut ↑ := *Stmt*.symTab ↓ . . .]
| *DeclType Var* = **new** *NewType* (*ArgList*)
[*Stmt*.symTabOut ↑ := *Stmt*.symTab ↓ +
(*Var*.name ↑ ↦ *DeclType*.type ↑) . . .]

(d)

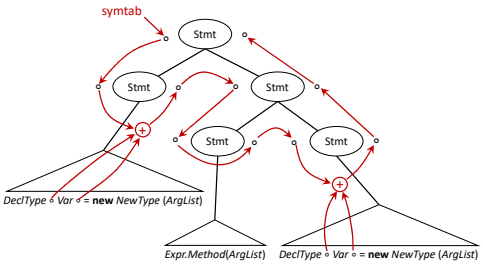

Figure 2: (a) A basic context-free grammar. (b) Attributes of the *Stmt* nonterminal. (c) Attribute equations for the productions (the parts of the equations denoted by "...." are elided). (d) An attributed tree, illustrating left-to-right threading of attributes.

Example. Consider the simple CFG in Fig. 2-(a). The nonterminal *Stmt* stands for *program statements*. The grammar says that a statement can either be a method call or a variable declaration, or a sequential composition of statements. A natural AG extension of this CFG tracks symbol tables, which allow easy lookup of all variables in scope.

Specifically, the grammar associates two symbol-table-valued attributes, symTab ↓ and symTabOut ↑, with *Stmt* (Fig. 2-(b)). The attributes are propagated following the equations in Fig. 2-(c). In these equations, we distinguish between the three different occurrences of nonterminal "*Stmt*" via the symbols "*Stmt*\$0," "*Stmt*\$1," and "*Stmt*\$2." where the numbers denote the leftmost occurrence, the next-to-leftmost occurrence, etc. In this case, the leftmost occurrence is the left-hand-side occurrence.

For concreteness, let us consider the attribute equations for the production for sequential composition in the grammar. Here, the inherited attribute of *Stmt*\$0 gets passed "down" the syntax tree as an inherited attribute of *Stmt*\$1. The synthesized attribute received at *Stmt*\$1 is passed to *Stmt*\$2 as an inherited attribute. More generally, the attribute equations define a left-to-right information flow through the syntax tree, as illustrated in Fig. 2-(d).

4 Neurosymbolic Attribute Grammars

Now we introduce the model of neurosymbolic attribute grammars (NSGs). Our goal is to learn a distribution $\mathcal{P}(Y|\mathsf{X})$, where Y is a random variable whose domain is all possible programs (concretely, Java method bodies) and X is a specification of a program-generation problem (concretely, an *evidence set* made up of useful information extracted symbolically from the method's context and then encoded using a neural network). Attributes containing the results of a symbolic, static analysis are available to the neural network implementing this distribution. This weak supervision allows the network to mimic more accurately the long-range dependencies present in real code-bases.

The Underlying Model. The idea of weak supervision using a static analyzer could be developed on top of many different kinds of generative models of code. Here, we develop the idea on top of a model introduced by Murali et al. [21]. This model uses a latent variable Z that models the *true user intent* behind the specification Y, which is assumed to be incomplete or ambiguous. We have: $\mathcal{P}(\mathsf{Y}|\mathsf{X}) = \int_{\mathsf{Z}} \mathcal{P}(\mathsf{Z}|\mathsf{X})\mathcal{P}(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}$. To define the distribution $\mathcal{P}(Z|\mathsf{X})$, we assume that the evidence set is divided into data of a finite number of *types*, e.g., method names, formal parameter types, and Javadoc comments that are extracted from a static analysis of the program. The jth type of evidence has a neural encoder f_j, and we assume that each piece of evidence X can either be represented as a vector (a JavaDoc comment encoded using a bi-directional RNN, for example), or as a set of encoded items with no particular ordering (a set of formal parameters, each encoded as a vector). Let $\mathsf{X}_{j,k}$ refer to the kth instance of the jth kind of evidence in X. Assume a Normal prior on Z, and let $\mathcal{P}(\mathsf{X}|\mathsf{Z}) = \prod_{j,k} \mathcal{N}\left(f_j(\mathsf{X}_{j,k}) \mid \mathsf{Z}, \mathbf{I}\sigma_j^2\right)$. Assume that the encoded version of each type of evidence is

sampled from a normal distribution centered at Z. If f is 1-1 and onto, it follows that [21]:

$$\mathcal{P}(\mathsf{Z}|\mathsf{X}) = \mathcal{N}\left(\mathsf{Z} \mid \frac{\sum_{j,k} \sigma_j^{-2} f_j(\mathsf{X}_{j,k})}{1 + \sum_j |\mathsf{X}_j| \sigma_j^{-2}}, \frac{1}{1 + \sum_j |\mathsf{X}_j| \sigma_j^{-2}} \mathbf{I}\right)$$

Next, define the distribution $\mathcal{P}(Y|\mathsf{Z})$. To do this, consider a stochastic CFG which assumes (1) that a leftmost derivation is carried out, and (2) the probability distribution governing the expansion of a symbol in the grammar takes into account the sequence of all expansions so far, as well as an input value Z upon which all expansions are conditioned.

This CFG consists of productions of the form $S : seq_1 \mid seq_2 \mid seq_3 ... \mid seq_n$. Each symbol such as S corresponds to a categorical random variable with sample space $\Omega(S) = \{seq_1, seq_2, ..., seq_n\}$. A trial over the symbol S randomly selects one of the RHS sequences for that symbol. If S is a terminal symbol, then $\Omega(S) = \{\epsilon\}$, where ϵ is a special value that cannot be expanded. Subsequently, when a trial over S is performed and an RHS sequence from $\Omega(S)$ is randomly selected, we will use the sans-serif S^{rhs} to denote the identity of the RHS sequence observed.

Now consider a depth-first, left-to-right algorithm for non-deterministically expanding rules in the grammar to generate a *program* $\mathsf{Y} = \langle (S_1, \mathsf{S}_1^{\text{rhs}}), (S_2, \mathsf{S}_2^{\text{rhs}}), ... \rangle$; here, each S_i is a symbol encountered during the expansion, and each $\mathsf{S}_i^{\text{rhs}}$ is the identity of the RHS chosen for that symbol. Let S_1 correspond to the symbol Start. We perform a trial over S_1 and select one of the RHS sequences from $\Omega(S_1)$. Let the identity of the RHS sequence selected be $\mathsf{S}_1^{\text{rhs}}$. Note that $\mathsf{S}_1^{\text{rhs}}$ is *itself* a sequence of symbols. Choose the first symbol in the sequence $\mathsf{S}_1^{\text{rhs}}$; call this symbol S_2. Perform a trial over S_2, and let the identity of the RHS sequence chosen be $\mathsf{S}_2^{\text{rhs}}$. Choose the first symbol in $\mathsf{S}_2^{\text{rhs}}$ (call it S_3) and expand it the same way. This recursive descent continues until a terminal symbol S_i is encountered, and the recursion unwinds. If the recursion unwinds to symbol S_2, for example, then we choose the second symbol in the sequence $\mathsf{S}_1^{\text{rhs}}$, which we call S_{i+1}. We perform a trial over S_{i+1}, and let the identity of the RHS sequence chosen be $\mathsf{S}_{i+1}^{\text{rhs}}$. This sequence is recursively expanded. Once all of the symbols in the RHS associated with the Start symbol S_1 have been fully expanded, we have a program.

Algorithm 1: Gen$(S, \downarrow A(S), \mathsf{SymSoFar}, \mathsf{Z})$

Input: current symbol S, inherited atts $\downarrow A(S)$, sequence of symbols so far $\mathsf{SymSoFar}$, latent pos Z

Modifies: all symbols expanded are appended to $\mathsf{SymSoFar}$

Returns: $\uparrow A(S)$, the synthesized atts of S

```
if S is a terminal symbol then
    Append (S, ε) to SymSoFar
    return ∅
else
    Choose a right-hand-side (RHS) sequence
     S^rhs ~ P(S|SymSoFar, ↓A(S), Z)
    Append (S, S^rhs) to SymSoFar
    SynthSoFar ← ⟨⟩
    for S' ∈ S^rhs in left-to-right order do
        Compute ↓A(S') from ↓A(S) and
         SynthSoFar
        ↑A(S') ←
         Gen(S', ↓A(S'), SymSoFar, Z)
        Append ↑A(S') to SynthSoFar
    end
end
Compute ↑A(S) from ↓A(S) and SynthSoFar
return ↑A(S)
```

This generative process defines a probability distribution $\mathcal{P}(Y|\mathsf{Z})$, where for a particular program Y, the probability of observing Y is computed as

$$\mathcal{P}(\mathsf{Y}|\mathsf{Z}) = \prod_i \mathcal{P}(S_i = \mathsf{S}_i^{\text{rhs}} | S_1 = \mathsf{S}_1^{\text{rhs}}, ..., S_{i-1} = \mathsf{S}_{i-1}^{\text{rhs}}, \mathsf{Z}). \tag{1}$$

We henceforth abbreviate the expression for the inner probability as $\mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \mathsf{S}_1^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}}, \mathsf{Z})$.

Weak Supervision with Attributes. Now assume that the grammar is an AG, so that each symbol S has an attribute set $A(S)$. We use $\uparrow A(S)$ to denote the synthesized attributes of S, and $\downarrow A(S)$ to denote the inherited attributes of S.

An NSG extends the model so that the conditional distribution $\mathcal{P}(Y|\mathsf{Z})$ is defined as:

$$\mathcal{P}(\mathsf{Y}|\mathsf{Z}) = \prod_i \mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \langle \mathsf{S}_1^{\text{rhs}}, \mathsf{S}_2^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}} \rangle, \downarrow A(S_i), \mathsf{Z}).$$

That is, when a symbol S_i is non-deterministically expanded, its value depends not just on the latent position Z and the sequence of expansions thus far, but also on the values of S_i's inherited attributes, $\downarrow A(S_i)$. In theory, a powerful enough learner with enough data could learn the importance of these sets of attribute values, without ever seeing them explicitly. In that sense, they could be treated as latent variables to be learned. However, the benefit of having a static analysis produce these values deterministically is the author of a static analysis knows the semantic rules that must be followed by a program; by presenting the data used to check whether those rules are followed directly to a learner, the process of learning to generate programs is made much easier.

Generation of a program under an NSG is described in Algorithm 1, where the distribution governing the expansion of symbol S has access to attribute values $\downarrow A(S)$.

Designing an appropriate static analysis. Intuitively, a program generated with the supervision of a static analyzer is likely to generate a semantically correct program because the static analysis provides key semantic clues during program generation. In a conventional AG-based analyzer, the AG would be used to maintain data structures that can be used to check, in an existing program, whether key relationships hold among the values of the production's attributes. Our goal is to generate and not validate programs. However, thinking of constraints provides a good mental model for designing a static analysis that can provide a neural network with the clues necessary to generate a correct program. That is, we generally want the attribute equations of the AG to be written so that if we did use constraints, the attributes necessary to check those constraints would be present where crucial choices will be made in the expansion. These sites are typically at the leaves of the grammar, where particular variables, types, and method calls are actually selected. An example of NSG design is in the supplementary material.

5 Evaluation

Our hypothesis is: it is difficult for a fully neural model to learn the intricate rules (such as typing and scoping rules) that govern the generation of code, by simply looking at a large number of example codes. This makes conditional program generation (CPG) difficult, especially if the program units to be generated are large, such as entire method bodies. However, a NSG can learn to use the result of a static analysis that is explicitly presented to it in the form of a symbol table and other data structures, to generate accurate code.

Data Considered. To test this hypothesis, we use a highly-curated, de-duplicated set of Java source-code files [21]. For each class and each method, we use the remainder of the class as evidence or context, and the method body is used to produce training or test data. We use 1.57M method bodies for training. The grammar used had ten terminals corresponding to formal parameters, ten for class variables, and ten for methods local to the class. None of the Java classes in the corpus needed more than ten of each of these terminals; when generating training data, each declared Java variable or method was randomly mapped to one of the appropriate terminals. Approximately 8,000 types and 27,000 method calls from the Java JDK also appeared as terminals in the grammar.

NSG implementation. We implemented an NSG for our subset of Java, where attributes are used to keep track of the state of the symbol table, the expected return type of each method, expected types of actual parameters, variable initialization, whether the variable has been used, and whether the method has a return statement. The symbol table contains entries for all formal parameters, class variables, and internal methods within the class. We use the static analyzer described in Sec. 4 to extract attributes from an input program.

To expose the attributes to the neural part of the model, we implement a depth first search over a program's abstract syntax tree (AST) to extract node information. The attributes are then encoded in a standard way — for example, the symbol table is represented as matrix (rows correspond to types, columns to variables, the value 1 is present if the corresponding type/variable pair is in scope). The distribution $\mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \langle \mathsf{S}_1^{\text{rhs}}, \mathsf{S}_2^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}} \rangle, \downarrow A(S_i), \mathsf{Z})$ is implemented as a set of LSTMs that decode the sequence of symbols as well as the encoded $\downarrow A(S_i)$ and Z into a distribution over $\mathsf{S}_i^{\text{rhs}}$. We trained our framework on top of Tensorflow [1]. Using one GPU, the NSG training time is around 72 hours. More implementation details are in the Appendix D.

Baselines. We consider three baselines. The first of these is a "conditional neural grammar" or CNG for Java, which is identical to our NSG model, but trained without access to $\downarrow A(S_i)$. The

Table 1: Percent of Static Checks Passed

	50% Evidence Visible				100% Evidence Visible			
	NSG	CODEGPT	GNN2NAG	CNG	NSG	CODEGPT	GNN2NAG	CNG
No undeclared variable access	**99.18%**	24.74%	47.17%	18.31%	**99.37%**	85.33%	47.44%	19.78%
Valid formal param access	**98.01%**	NA	25.50%	10.45%	**98.16%**	NA	25.78%	11.03%
Valid class var access	**99.07%**	NA	14.96%	12.52%	**99.37%**	NA	15.40%	12.75%
No uninitialized Objects	**88.08%**	33.55%	20.01%	18.74%	87.21%	**92.48**	21.20%	21.56%
No variable access error	**98.99%**	27.03%	28.43%	17.00%	**99.16%**	88.71	28.92%	17.92%
Object-method compatibility	**86.20%**	81.82%	21.39%	10.36%	**91.77%**	46.95%	21.43%	12.23%
Ret Type at call site	**88.27%**	23.19%	23.45%	12.27%	**90.57%**	87.50%	23.86%	16.40%
Actual Param Type	**96.76%**	1.29%	9.24%	17.97%	**97.90%**	1.79%	9.27%	16.09%
Return Stmt Type	**79.39%**	1.00%	12.07%	11.74%	**88.70%**	9.20%	12.34%	9.51%
No Type Errors	**88.03%**	28.74%	20.04%	16.09%	**92.37%**	50.42%	16.31%	13.56%
Return Stmt Exists	99.00%	99.50%	93.87%	**99.64%**	99.09%	85.78%	94.02%	**99.92%**
No Unused Variables	84.88%	**92.31%**	20.55%	22.60%	87.22%	**94.66%**	20.95%	24.29%

Table 2: Average Fidelity of Generated Method Bodies

	50%Evidence Visible				100%Evidence Visible			
	NSG	CODEGPT	GNN2NAG	CNG	NSG	CODEGPT	GNN2NAG	CNG
Set of API Calls	**38.54**	0.00	2.03	12.42	**43.68**	4.67	2.21	22.33
Sequences of API Calls	**29.00**	0.00	0.22	7.50	**34.04**	2.00	0.29	18.50
Sequences of Program Paths	**26.01**	0.00	0.00	7.95	**31.32**	2.00	0.00	17.45
AST Exact Match	**14.29**	0.00	0.00	1.00	**18.95**	0.00	0.00	6.00

second is a pre-trained CODEGPT model Lu et al. [19] fine-tuned on our Java dataset. The third is GNN2NAG [7], a graph neural network based method that uses an attribute grammar, but which does not use an explicit static analysis. See Appendix D for more details on the last two baselines.

Test Scenario. Given a class, we remove a random method body to simulate a scenario in which the user is in the process of developing that class, and has decided to use CPG to aide in writing a method body. We use two settings for evidence visibility during CPG: 50% and 100% ($X\%$ means that $X\%$ of the evidence from the class, aside from the missing method body, is visible to the method). For each of 1000 test methods, we use beam search with a beam width of ten to obtain the ten method bodies with highest likelihood.

5.1 Quantitative Results

Static checks considered. For each generated method body, we check: *(1) No undeclared-variable access:* are all the variables used in a program declared before they are used, and are they in scope? *(2) Valid formal-parameter access:* are formal parameters from the grammar that are used in the method body present in the method declaration? *(3) Valid class-variable access:* are class variables from the grammar that are used in the method body present in the class declaration? Note that for all methods other than CODEGPT, our Java parser partitions variable tokens into different types (class variables, formal parameters, etc.). (2) and (3) check whether these tokens are used correctly. Because CODEGPT does not use a formal grammar, (2) and (3) are not meaningful metrics. *(4) No uninitialized objects:* do variables have a non-null value when they are used? *(5) No variable-access errors:* are checks (1)-(4) satisfied? *(6) Object-method compatibility:* are methods called on objects that have that method type available? *(7) Return type at the call site:* is the assignment of the return value type-correct with respect to the return type of the called method? *(8) Actual-parameter type:* are the actual-parameter types in an API call consistent with the corresponding formal-parameter types? *(9) Return-statement type:* is the type of the expression in a return statement consistent with the method's declared return type? *(10) No type errors:* are checks (6)-(10) satisfied? *(11) Return statement exists:* does the method body have a return statement somewhere? *(12) No unused variables:* what fraction of variables declared in the method body are used? Results are shown in Table 1. These scores are interpreted as follows. Suppose that the generated program uses five variables, of which four are declared correctly in the proper scope. This is scored as 80% correct on the "No undeclared-variable access" test or as 80% correct. We report the average success rate over each of these semantic properties over all the synthesized programs in our test suite.

Testing whole-method fidelity. We also check fidelity of the generated code to the original code. One option is BLEU score, but this is problematic for evaluating method body generation. BLEU

score is not invariant to variable re-namings. Programs are structured, with control flow, and not sequential. Some tokens (such as `if`) indicate control flow, and some (object references) indicate data flow; treating these all equivalently (as words in machine translation) does not make sense.

	Reading from a file	Adding to a list

Query	`public class FileUtils{`
	`FileReader field_7;`
	`BufferedReader field_5;`
	
	`/** read line from file */`
	`public String reader () {???}}`
		`public class myClass{`
		
		`/** add item to list */`
		`public void addItem (`
		`List<String> a,`
		`String b){???}}`

NSG	`public String reader(){`
	`java.lang.String var_9;`
	`try {var_9=field_5.readLine();`
	`} catch (IOException var_8) {`
	`var_8.printStackTrace();`
	`}`
	`return var_9; }`
		`public void addItem (`
		`List<String> fp_9,`
		`String fp_1) {`
		`if (fp_9.contains`
		`(String: LITERAL)) {`
		`fp_9.add(fp_1);}`
		`return; }`

CODEGPT	`public String reader(){`
	`StringBuffer buffer=`
	`new StringBuffer();`
	`buffer.append("\n");`
	`return buffer.toString();`
	`}`
		`public void addItem (`
		`List<String> a,`
		`String b) {`
		`items.add (a);}`

Table 3: Example synthesis outputs.

As such, we consider four fidelity metrics. (1) extract the set of Java API calls from the generated and reference codes, and check the sets for Jaccard similarity; (2) use static analysis to generate the set of all possible API call sequences possible along code paths, and check the Jaccard similarity across generated and reference code; (3) generate the set of all possible paths from root to leaf in the AST, check for Jaccard similarity (two paths are equal if tokens except for object references match); (4) exact AST match (except for object references), which evaluates to 0 or 1. We compute the highest value for each metric across the ten bodies generated, and average the highest across all test programs. The results are presented in Table 2.

Discussion. We find that in most cases, the NSG had a significantly higher incidence of passing the various static checks compared to the baselines. This is perhaps not surprising, given that the NSG has access to the result of the static analysis via the attribute grammar. More intriguing is the much higher accuracy of the NSG for the fidelity results. Both CODEGPT and GNN2NAG are designed for next-token-prediction tasks (we give some results on these tasks in Appendix H). However, in our CPG task, no tokens are available from the method body to be generated. CODEGPT and GNN2NAG must treat the surrounding code and method header as input from which to generate the method body, and how to process this complex context to "figure out" what body to produce. This proves difficult — for example, CODEGPT cannot generate any programs matching the ground truth when 50% evidence is visible. The NSG, on the other hand, uses a static analysis to symbolically extract this context, which is explicitly given to the neural network in the form of the class variables and methods which are available to be called (in $\downarrow A(S)$) and in the class name, encoded comments, variable names, and so in, (in Z).

5.2 Qualitative Results

We now consider some example codes produced by the NSG proposed in this paper and CODEGPT, as shown in Table 3 and Fig. 3. Additional (and longer) examples appear in the Appendix of the paper. Consider the task of generating a code to read from a file (depicted in Table 3). The NSG is able to correctly generate a code that uses the `BufferedReader` (referred to by the member variable `field_5`) to read into a `String` object, and return that object to the caller. In contrast, CODEGPT inexplicably creates a `StringBuffer`, appends a newline to it, and returns that newline. Or, compare the CODEGPT-generated code in Fig. 3 with the NSG-generated code in Fig. 1. The NSG-generated code has already been discussed and is likely intuitive to most readers, but the CODEGPT-generated code simply prints the `File` object to the screen, as opposed to actually attempting to write the contents of the `String` to the `File` object that is passed in.

```
void write (File a, String b){
System.out.println(a);}
```

Figure 3: CODEGPT-generated code for the task of Figure 1.

6 Related Work

Statistical Models of Code. Many non-neural models of code have been proposed over the years [20, 26, 23, 3, 22, 6]. A few of these models condition generation on symbolic information from the context. Specifically, Bielik et al. [6] use programmatically represented functions to gather information about the context in which productions for program generation are fired, then utilize this information to impose a distribution on rules. Maddison & Tarlow [20] generate programs using a model that encodes a production's context using a set of "traversal variables". However, the absence of neural representations in these models puts a ceiling on their performance.

There is, by now, a substantial literature on deep models trained on program syntax. Early work on this topic represented programs as sequences [26] and trees [21, 31, 8] and learned using classic neural models such as RNNs, as well as specialized architectures [18, 24, 4]. The recent trend is to use transformers [29, 14, 13, 19]. These methods do well on short-horizon code completion tasks. However, our experiments with CODEGPT suggest that they have difficulty respecting the semantics of the language as the task horizon lengthens.

The most closely related work to ours is by Brockschmidt et al. [7], who construct graph representations of the context as generation progresses and use a neural model (GNNs) to construct attributes from these graphs. The benefit of our approach over this method is that we do not need to learn the attributes and instead construct them symbolically. Our experiments demonstrate the value of this strategy on long-horizon tasks.

Also related is work by Dai et al. [10], who extend grammar variational autoencoders [17] with hard constraints represented as attribute grammars. In that work, attribute constraints are propagated top-down, and every generated artifact is required to satisfy the top-level constraint. This strategy comes with challenges; as is well-known in the program-synthesis literature [25], top-down constraint propagation can lead to unsatisfiability, and require rejection of generated samples, for grammars above a certain level of complexity. We sidestep this issue by using attribute grammars as a form of weak supervision, rather than as a means to enforce hard constraints.

Neurally Directed Program Synthesis. Conditional program generation is closely related to *program synthesis*, the problem of searching for programs in domain-specific languages (DSLs) that implement a user-given constraint. A body of recent work uses neural techniques [5, 11, 27, 9] to address this problem. The first difference between this setting and ours is that we do not aim for a complete search over programs; our decoder performs a beam search the width of this beam is limited. Also, while we try to satisfy *language-level* constraints, we do not allow users to specify additional hard constraints during generation. These restrictions allow us to go beyond contained DSLs and handle a general-purpose programming language.

7 Conclusion

We have presented a framework for deep generation of source code in which the training procedure is weakly supervised by a static analyzer, in particular, an attribute grammar. Our implementation of the approach outperforms a state-of-the-art transformer, as well as a method that tries to explicitly learn program semantics, on long-horizon conditional program generation tasks.

The greatest limitation of our approach is that while the idea of program generation modulo static analysis is language-agnostic, implementations of the idea are not. To instantiate our framework for a new language, one must write a new static analyzer, which can be costly. In contrast, models such as CodeGPT can trivially accommodate training data from a new language. Also, our current implementation of NSGs is built on top of tree LSTMs as opposed to the more contemporary transformer architecture. This is for good reason, as transformers view programs as sequences of tokens, while static analysis techniques operate on more abstract, tree-structured models of code. However, the use of LSTMs comes with a performance penalty, and more research needs to be done on integrating static-analysis-based weak supervision with transformers. Finally, any implementation of our approach can only offer supervision regarding a bounded number of program properties. The extent to which the code the system generates will respect properties outside this set is unclear.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[3] Allamanis, M. and Sutton, C. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635901. URL http://doi.acm.org/10.1145/2635868.2635901.

[4] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[5] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[6] Bielik, P., Raychev, V., and Vechev, M. PHOG: Probabilistic model for code. In *ICML*, pp. 19–24, 2016.

[7] Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

[8] Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *arXiv preprint arXiv:1802.03691*, 2018.

[9] Chen, Y., Wang, C., Bastani, O., Dillig, I., and Feng, Y. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pp. 587–610. Springer, 2020.

[10] Dai, H., Tian, Y., Dai, B., Skiena, S., and Song, L. Syntax-directed variational autoencoder for structured data. *arXiv preprint arXiv:1802.08786*, 2018.

[11] Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

[12] Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

[13] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[14] Gemmell, C., Rossetto, F., and Dalton, J. Relevance transformer: Generating concise code snippets with relevance feedback. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2005–2008, 2020.

[15] Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[16] Knuth, D. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[17] Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. Grammar variational autoencoder. In *International Conference on Machine Learning*, pp. 1945–1954. PMLR, 2017.

[18] Ling, W., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., Senior, A., Wang, F., and Blunsom, P. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[19] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[20] Maddison, C. and Tarlow, D. Structured generative models of natural source code. In *ICML*, 2014.

[21] Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. In *ICLR*, 2018.

[22] Nguyen, A. T. and Nguyen, T. N. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 858–868, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818858.

[23] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 532–542, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491458. URL http://doi.acm.org/10.1145/2491411.2491458.

[24] Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[25] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[26] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *PLDI*, 2014.

[27] Si, X., Yang, Y., Dai, H., Naik, M., and Song, L. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.

[28] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[29] Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

[30] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

[31] Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pp. 440–450, 2017.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.

- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 7
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] We have proposed an approach to better generation of source code and used open-source programs as training data. We do not anticipate any direct negative societal impact of this work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See Supplementary material
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Supplementary material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] We reported the mean due to space limit, the standard deviation is small enough to be negligible.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 5
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] See Supplementary material
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]