Next Syntactic-Unit Code Completion and Applications

Anh Tuan Nguyen
Axon
Hanoi, Vietnam
ntanhbk44@gmail.com

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Code completion is an important feature in an IDE to improve developers' productivity. Existing code completion approaches focus on completing the current code token, next token or statement, or code pattern. We propose AstCC, a code completion approach to suggest the next syntactic unit via an AST-based statistical language model. AstCC learns from a large code corpus to derive the next AST subtree representing a syntactic unit, and then fills in the template with the concrete variables from the current program scope. Our empirical evaluation shows that AstCC can correctly suggest the next syntactic unit in 33% of the cases, and in 62% of the cases, it correctly suggests within five candidates. We will also explain the potential applications of AstCC in automated program repair, automated test case generation, and syntactic pattern mining.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Code Completion; Statistical Language Model

ACM Reference Format:
Anh Tuan Nguyen, Aashish Yadavally, and Tien N. Nguyen. 2022. Next Syntactic-Unit Code Completion and Applications. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3551349.3559544

1 INTRODUCTION

A code completion (CC) tool in an integrated development environment (IDE) (e.g., Eclipse [8], IntelliJ IDEA [9]) helps users speed up the coding task by filling the desired code and reducing common mistakes. Code completion is *the third most useful feature in an IDE* (behind editing and compiling) [12]. Basic CC helps users complete the names of classes, methods, fields, and keywords within the visibility scope [9]. If CC is applied to a part of a field, parameter, or variable declaration, the tool suggests a list of possible names according to the entity's type. Advanced CC tools [9] support the completion in common program constructs, the right part of assignments, variable initializers, arguments of a method call, etc.

A code completion engine, as regarded as general code synthesis, is actually useful in the other tasks, e.g., automated program repair [11], test generation in automated testing [16], code synthesis [5, 17], etc. A key common requirement for CC in those tasks is *the ability to synthesize any syntactic unit at any location*. However, the majority of the CC approaches support *next-token* completion, and very few of them support *next-statement* completion [15].

Those CC services are built mainly using *program analysis* (PA) on the currently edited code in the IDE. The issue with PA-based CC approaches is that there is a large amount of possibilities with equal likelihoods of code candidates that can be filled in at the cursor. For example, after `len =`, the valid next-token candidates include a limited set of code tokens. However, there is potentially an infinite number of valid statements at that point. Moreover, PA cannot rank the candidates based on their occurrence likelihoods.

To address those issues, several *code mining* and *information retrieval* CC approaches have been proposed [1, 14, 18]. The idea is that the CC tool could suggest and rank higher the more frequent code. However, the tokens to be filled for the current code might not be as frequent, leading to ineffectiveness of those approaches. Moreover, code is repetitive as single tokens; while as entire statements, it is quite specific for projects [19]. Thus, searching/mining for a similar code to be filled is not effective across projects.

Deep learning (DL) and *natural language processing* (NLP) have enabled several approaches for code completion using *statistical language models (LM)* and *DL models*, which capture the regularity of source code [3, 4, 7, 10, 15]. However, these approaches do not explicitly account for the syntactic correctness of generated code. Moreover, Chen *et al.* [2] qualitatively investigate Codex for varied levels of abstraction to find that it can recommend syntactically incorrect or undefined code; and can invoke functions, variables, and attributes that are undefined or outside the scope of the codebase.

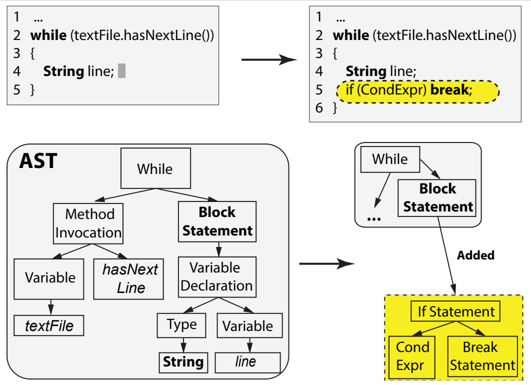
2 COMPLETING NEXT SYNTACTIC UNIT

We propose AstCC that supports the suggestion of the next syntactic unit. The most likely syntactic template is first suggested and then filled with the proper program elements in the current scope. AstCC engine is general to work for both interaction within an IDE or a setting of code synthesis at the current location.

AstCC focuses on next syntactic unit CC that (1) has not been investigated much in the literature, and (2) can be used to improve the existing techniques in code completion and other applications.

Let us start with an example (Figure 1). A user is currently editing a `while` loop in which (s)he would like to iterate over all the text lines in a file. AstCC will suggest the addition of the `if` statement with a condition expression and a `break` statement. The reason is that it has often encountered the templates where some condition is evaluated inside a while loop. If the condition is satisfied, the execution will break out of the loop. Our model learns from a large

Figure 1 Suggesting a Valid Syntactic Unit



```
...
while (textFile.hasNextLine())
{
   String line;
}
```

```
...
while (textFile.hasNextLine())
{
   String line;
   if (CondExpr) break;
}
```

code corpus such common syntactic structure, i.e., a while loop with an if-break. Let us call it a *syntactic template*. AstCC will concretize the template to complete that syntactic unit.

To realize AstCC, we follow the direction of an AST-based statistical language model (LM). There are two key departure points in AstCC in comparison with existing CC approaches.

1) We do not predict the next token based on the sequence of previous tokens in the current code. Instead, we leverage AstLan [13], an AST-based LM that *predicts the next "expansion" of the current AST subtree* based on the surrounding code structure.

2) Instead of verifying the syntactic correctness of candidate code after generating them, we *integrate such verification into the process of learning the expansion from a smaller AST subtree to a larger one* (Section 4). With this integration strategy, our model learns the valid expansions, i.e., syntactically correct code, thus, ensuring that the suggested next syntactic units will be valid. The choice of using statistical LM (rather than Deep Learning) makes AstCC more light-weight and facilitates the above integration. Except a few high-computational models [2–4], most DL-based CC models focus on *next-token* [10] or *next-statement* [15] completion.

3 ASTLAN: AST-BASED LANGUAGE MODEL

Ancestor-Descendant AST relation. *The key idea is that the next syntactic unit is the addition of some AST subtree to the current AST.* As shown in Figure 1, the newly suggested syntactic unit (i.e., the if statement with break) is the extension/addition to the current AST on the left. Therefore, AstLan models the expansion from a smaller AST to a large one, instead of expansion from a smaller sequence to a large one as in a sequence-based language model (e.g., *n*-gram). The relation between a smaller AST and the larger one with additional nodes is called *ancestor-descendant relation.*

Definition 1 (**Ancestor-Descendant ASTs [13]**). An AST C is a **descendant** of AST P (P is a **ancestor** of C) if
1) C is formed by adding a minimal (sub)tree T to a node in P; and
2) P and C are syntactically valid.

A minimal T is defined as the subtree such that if any node in T is removed, C will become syntactically incorrect. The *first condition* is aimed to produce the next smallest, valid AST template of a certain syntactic type. This is enabled via the descendant AST C in the Ancestor-Descendant relation, such that the descendant must not contain another valid syntactic type that is larger than the ancestor AST P, and smaller than the descendant AST C. This can only be achieved by ensuring that the subtree T that is to be added to P is minimal/irreducible among the subtrees with the same syntactic type. The IfStatement subtree added as a child node of BlockStatement in Figure 1 (highlighted in yellow), is the smallest IfStatement subtree that produces a syntactically valid tree. The *second condition* ensures the validity of the suggested code. In Figure 1, the suggested subtree at the IfStatement is syntactically valid. Any added smaller subtree, such as IfStatement $\longrightarrow$ Cond, would produce invalid syntactic code (missing the True part).

To enhance the capability of applying the template learned from one place to another, we alpha-rename the variables and keep only the token type and the data type for each AST node corresponding to an identifier. This allows for bound variable names to be changed. For a local variable node in a subtree or a label in a switch statement, its label is the name of that variable (via alpha-renaming within the subtree), concatenated with its type [13]. In Figure 1, textFile becomes VAR1_Scanner, and line becomes VAR2_String. A literal becomes a label 'LIT' with its type attached at the end. The special values such as empty string, zero, and null are abstracted with special string tokens, indicating special situations, e.g., nullity checking.

4 ASTCC: NEXT SYNTACTIC UNIT SYNTHESIS

Step 1. Training for AstLan. The first important step is to train the model by mining all the pairs of ancestor-descendant ASTs from a corpus of syntactically correct programs. Because AstLan is a general language model, it does not consider the syntactic correctness of the generated code. Thus, to ensure that correctness, in AstCC, we modify the training process in AstLan as follows.

Given a method, we build its AST. We traverse the AST from the top and identify the pairs of ancestor and corresponding descendent ASTs. When encountering an ancestor tree, we expand to visit a descendant tree according to the rules in Table 1.

First, our algorithm aims to find one or more valid AST subtrees as initial ancestor AST(s). Depending on the type of the current AST node n, AstCC expands the current subtree to include one of its children nodes to form with n, a syntactically correct tree. Let us take an example of the rule for IfStatement. There are three valid expansions: 1) connecting if to E and S1 (i.e., the true branch), and 2) connecting if to E and both branches S1 and S2. We cannot connect if to only the S2 branch because it would create an invalid code. The presence of the true branch is necessary. After an expansion, we consider one of the children nodes and repeat the same expansion with those rules until it hits a leaf node. At an expansion step, for a possibility, after traversing to c's children, if the resulting AST fragment formed by the tree expanding to c, c itself, and c's children, is valid, we will consider it as a descendant tree. For example, in Figure 1, we have two initial ancestor ASTs: 1) the left subtree at While (P_1), and 2) left subtree at While and BlockStatement (P_2).

Let us detail the process of expansion via the edges coming out of the initial ancestor tree P. Let us use n_i to denote the node at the end of an outgoing edge from the ancestor tree P. For example, BlockStatement is such a node for P_1 and VariableDeclaration is such a node for P_2. To find a descendant tree of the ancestor tree, we

Table 1: Examples of Expansion Rules [13]

Syntax	**Valid Expansion**
If ::= **if** E S1 S2	If ⟶ E, S1
	If ⟶ E, S1, S2
While ::= **while** E Stmt	While ⟶ E
	While ⟶ E, Stmt
For ::= **for** Init E Update Stmt	For ⟶ Init, E, Update
	For ⟶ Init, E, Update, Stmt
Switch ::= **switch** E Case* Def	Switch ⟶ E
	Switch ⟶ E, F with F ∈ all Case combinations
	Switch ⟶ E, Def
	Switch ⟶ E, F, Def with F ∈ all Case combs
Case ::= **case** E: Stmt	Case ⟶ E
	Case ⟶ E, Stmt
InfixOp ::= E1 Op E2	InfixOp ⟶ E1, E2
EnhancedFor ::= VarDec, Ref, Stmt	ForEach ⟶ VarDec, Ref
	ForEach ⟶ VarDec, Ref, Stmt
Do ::= Stmt, Cond	Do ⟶ Stmt, Cond
Try ::= **try** Block {Catches | Finally}	Try ⟶ Block, all combinations of Catches
	Try ⟶ Block, Finally
	Try ⟶ Block , all comb. of Catches, Finally
Conditional ::= E1 ? E2 : E3	Conditional ⟶ E1, E2, E3
Synchronized ::= Exp, Stmt	Synchronized ⟶ Exp, Stmt
Labeled ::= Lit, Stmt	Labeled ⟶ Lit, Stmt
Variable Dec. ::= TypeRef, VarSpec	VarDec ⟶ TypeRef, VarSpec
Variable Spec. ::= Name, Init	VarSpec ⟶ Name
	VarSpec ⟶ Name, Init
Type Reference ::= TypeName, TypeArg	TypeRef ⟶ TypeName
	TypeRef ⟶ TypeName, TypeArg
Other	All combinations

attempt to expand from the ancestor to include n_i and further to n_i's children node. The expansion rules in Table 1 are used to determine the valid edges for expansion. The node n_i and each of the valid combinations of its children nodes to form different possible subtree(s) are collected. Among them, the subtrees with the minimum number of nodes is used to connect to the ancestor tree to form one of the descendant trees for the ancestor tree.

The possible subtrees with larger sizes are used as the further descendant ASTs of the collected descendant ASTs when the expansion continues. At the later step, the descendant ASTs are used as the ancestor ASTs for the next traversal. For example, P_1 is an ancestor AST of P_2, which in turn is an ancestor AST of the entire subtree at While. To derive other ancestor AST for a descendant AST C, each ancestor of P is connected to the corresponding T of C. If the resulting tree is valid and connected to the ancestor AST, we consider it as a new ancestor of C. The training data consists of all the mined ancestor and descendant AST sub-trees.

Step 2. Predicting/Generating the template of the next valid AST subtree. This section explains how we use the collection of ancestor-descendant ASTs to suggest the complete valid syntactic template. AstCC relies on the Bayesian Statistical Inference with the input from the subtrees in the current code as the context. AstLan performs the prediction as follows. First, AstLan extracts the contextual information in the form of context trees from the current code. It finds the smallest, valid subtree T_S with the corresponding source code covering the current position. AstLan collects all the AST subtrees that contain the root of T_S and have a height greater than a certain threshold. Those subtrees are considered as the context for our inference to predict the next descendant AST. In Figure 1, if we extract the context from the subtree at BlockStatement, the trees rooted at WhileStatement and BlockStatement with the heights smaller than a threshold are collected in the context.

Algorithm 1 Concretizing Syntactic Template

```
1:  function Main($templ$, $V$)
2:      $candList = concretizeNext(templ, V, \emptyset, 1)$
3:      return $candList$

4:  function Concretize($templ$,$V$,$curCandList$,$loc$)
5:      if $loc > size(templ)$ then return $curCandList$
6:      $codeCands = \emptyset$
7:      $codeTokens = \alpha(templ[loc], V)$
8:      if $curCandList = \emptyset$ then
9:          for all $t \in codeTokens$ do
10:             $newCand = connect(EMPTY\_TREE, t)$
11:             $codeCands.adds(newCand)$
12:     else
13:         for all $t \in codeTokens$ do
14:             for all $cand \in curCandList$ do
15:                 $newCand = connect(cand, t)$
16:                 $codeCands.adds(newCand)$
17:     return $Concretize(templ, V, codeCands, DFS.next(loc))$
```

According to Nguyen *et al.* [13], given the context trees, the probability of a given tree T can be computed based on the numbers of ancestor-descendant ASTs. Using the Bayesian statistical inference, the generation probability of a new valid AST $C(t)$ given the context including t (e.g., $t = t_i$) can be computed as follows [13]:

$$Pr(C(t)|Ctxt) = Pr((t, N^+, E^+)|t_1, .., t_n)$$
$$= \frac{\#methods(t_1,C(t))+\alpha}{\#method(C(t))+\alpha.\#methods} \cdots \frac{\#methods(t_{(i-1)},C(t))+\alpha}{\#method(C(t))+\alpha.\#methods} \cdot \frac{\#methods(t,C(t))}{\#methods(t)} \cdot \frac{\#methods(t)}{\#methods} \cdots \frac{\#methods(t_n,C(t))+\alpha}{\#method(C(t))+\alpha.\#methods} \quad (1)$$

$Pr(C(t)|Ctxt)$ is computed for each context tree t_j. Finally, the corresponding additional AST's subtrees are computed and ranked by those probabilities.

Step 3. Concretizing the variables' names. After having a list of ranked candidate templates, we need to concretize the variables' names in a template with the accessible variables in the current scope. We then use a language model on the concrete code sequences corresponding to the concretized subtrees for ranking.

The key concretization steps are in Algorithm 1. Each AST node corresponding to a variable in the syntactic template is concretized first via the accessible variables in the current scope. The candidate variables must also conform to the type of the AST node. Function α checks those two conditions to return the code tokens. Initially, after applying α, we obtain the initial concretized AST (lines 9–11). We recursively expand the AST (line 17) with each of the candidate trees, and produce the larger concretized tree (lines 13–16).

After finishing the concretization process for all the candidates, we obtain the list of code sequences of the candidates. We then rank them by training an n-gram language model on the lexical code sequences extracted from a large code corpus. All the source files are tokenized into the sub-tokens using Hungarian or Camelcase conventions. The trained n-gram model is used to estimate the occurrence likelihood of the sequences produced by our model (not shown in Algorithm 1). Given the current code C, the likelihood of the candidate syntactic unit γ is: $\phi(serialize(connect(C, \gamma)))$, where C and γ are the lexical forms of C and γ, respectively. $connect$ is

Table 2: Data Collection

Total projects	1,000
Total classes	104,645
Total methods	638,293
Total SLOCs	7,144,198
Total valid AST's fragments	1,047,614,720
Total distinctive fragments	36,608,102
Total distinctive AST nodes	302,367

Table 3: Accuracy % in Next-Syntactic-Unit Suggestion

Top-1	Top-2	Top-3	Top-4	Top-5
33.2	**42.6**	**43.7**	**50.6**	**62.1**

the function to connect for expansion with the subtrees. *serialize* is the function to serialize the AST into a code sequence.

5 PRELIMINARY EMPIRICAL EVALUATION

5.1 Accuracy in Suggesting Next Syntactic Unit

5.1.1 Experimental Setting. We reused a large Java corpus (Table 2) that have been used in prior code completion work [13] (duplicated code was removed). The ASTs are extracted from the source code and a database of pairs of ancestor-descendant ASTs is built. In total, we have about 1,047 billion ASTs, of which 36M are distinct.

For experiments, we used the simulation process as in the prior work [13]. We began with the first valid syntactic unit, i.e., the first valid AST subtree. Initially, the current position of the cursor is placed at the code location corresponding to the right-most leaf node in the first valid AST subtree. We then obtain the context subtrees with the code tokens of their leaf nodes appearing prior to the current location according to the description in Section 4. We use the code completion engine to suggest the top-k valid syntactic units. To measure the performance, we compare the predicted syntactic unit against the actual next valid AST. If they match, we count it as a hit and otherwise, as a miss. We repeat the process by moving the current location to the end of the next syntactic unit until the entire method is scanned. Top-k accuracy is defined as the ratio between the number of hits over the total number of suggestions. For a dataset, we perform 10-fold cross validation. We split the training/test sets by project.

5.1.2 Experimental Results. As seen in Table 3, with a single suggestion, in 33.2% of the cases, our model correctly suggests the next syntactic unit. In 62% of the cases, it correctly suggests the next unit within five candidates. We also manually examined the suggested syntactic templates and found that the model is able to learn interesting patterns. For example, a common syntactic pattern is the for loop, in which the body evaluates a condition, and correspondingly breaks out of the loop body. Several other patterns that are specific to software libraries were also found, for example, a pattern of the initialization of a display in the SWT library.

5.2 Accuracy in Next-Statement Suggestion

Regardless of our model suggestion being the next valid syntactic unit, we aim to evaluate it in the next-statement suggestion setting. We compare our code completion engine with two state-of-the-art next-statement code completion approaches: PCC [19] and AutoSC [15]. We use the same process as in the experiment for next-valid-syntactic-unit suggestion, except that we move the current location to the end of the next statement (rather than at the end of the next syntactic unit) until the entire method is scanned.

Table 4: Accuracy % in Next-Statement Suggestion

	Top 1	Top 3	Top 6	Top 10
AutoSC [15]	20.3	28.5	32.0	42.2
PCC [19]	28.9	51.1	54.8	59.3
AstCC	35.1	59.0	67.8	80.7

As seen in Table 4, AstCC performs better than both the baselines. While AstCC *integrates the process of verifying syntactically correct units into the candidate generation* process, the other two approaches use program analysis to verify the candidates. Thus, the program analysis components of AutoSC, e.g., identifying the valid next-token candidates or type-checking, do not perform well.

5.3 Future Applications and Plan

5.3.1 Real-world Code Completion Benchmark and Human Studies. Currently, we evaluate AstCC with the assumption of the invocation of auto-completion at every syntactic unit (or statement). We plan to perform an evaluation on the real-world CC data introduced in [6]. We also plan to conduct human studies in a controlled setting for real-world developer trials to quantify AstCC's performance.

5.3.2 Syntactic Patterns Mining. With AstCC's syntactic templates, we can tailor it to mine syntactic patterns such as those involving common structures, e.g., if, for, while, etc., or API common usages. These patterns will be useful for CC tool builders and researchers.

5.3.3 Automated Program Repair. Training AstCC with the corpus of bug-fixing changes, we could use it to predict the syntactic unit at the fixing location. The syntactic correctness of the generated code reduces the need of post-processing as in current APR tools.

5.3.4 Using AstCC in Automated Unit Test Generation. With the knowledge discovered from the codebase on the test code patterns, AstCC will help developers to complete their test code in a higher volume, and also a more complete and correct manner.

6 CONCLUSION

We propose AstCC, a novel code completion approach that utilizes a novel direction in AST-based statistical language modeling. AstCC suggests the next syntactic unit by learning from a large code corpus, the potential expansion from a smaller AST subtree to a larger one. The second key novelty in AstCC is the integration of syntactic correctness verification into the process of learning the expansion, thus, making the model learn the valid expansion of the next unit. After deriving the next syntactic template, AstCC fills in the template candidates with the concrete variables from the current scope. Our empirical evaluation shows a promising result with top-1 accuracy of 33% and top-5 accuracy of 62%.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, and the US National Security Agency (NSA) grant NCAE-C-002-2021 on Cybersecurity Research Innovation.

REFERENCES

[1] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 213–222.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[3] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota. 5555. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 01 (nov 5555), 1–1. https://doi.org/10.1109/TSE.2021.3128234

[4] Copilot [n. d.]. Copilot. https://copilot.github.com/.

[5] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 416–432. https://doi.org/10.1145/2814270.2814295

[6] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-world Completions. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 960–970. https://doi.org/10.1109/ICSE.2019.00101

[7] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 837–847.

[8] Informer [n. d.]. Informer. http://javascript.software.informer.com/download-javascript-code-completion-tool-for-eclipse-plugin/.

[9] Intellisense [n. d.]. Intellisense. http://blogs.msdn.com/b/vcblog/archive/tags/intellisense/.

[10] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[11] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[12] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How Are Java Software Developers Using the Eclipse IDE? *IEEE Softw.* 23, 4 (July 2006), 76–83. https://doi.org/10.1109/MS.2006.105

[13] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 858–868.

[14] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 69–79. http://dl.acm.org/citation.cfm?id=2337223.2337232

[15] Son Nguyen, Hoan Nguyen, Ngoc Tran, Hieu Tran, and Tien N. Nguyen. 2019. Feature-Interaction Aware Configuration Prioritization for Configurable Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE Press, 489–501. https://doi.org/10.1109/ASE.2019.00053

[16] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA, 75–84.

[17] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, United States of America) *(ICSE 2016 - to appear)*.

[18] R. Robbes and M. Lanza. 2008. How Program History Can Improve Code Completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, 317–326.

[19] Yixiao Yang, Yu Jiang, Ming Gu, Jiaguang Sun, Jian Gao, and Han Liu. 2017. A Language Model for Statements of Software Code. In *Proceedings of the 32nd*

IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, 682–687.