Synthesizing Java Expressions from Free-Form Queries

Tihomir Gvero   Viktor Kuncak [*]

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

firstname.lastname@epfl.ch

Abstract

We present a new code assistance tool for integrated development environments. Our system accepts as input free-form queries containing a mixture of English and Java, and produces Java code expressions that take the query into account and respect syntax, types, and scoping rules of Java, as well as statistical usage patterns. In contrast to solutions based on code search, the results returned by our tool need not directly correspond to any previously seen code fragment. As part of our system we have constructed a probabilistic context free grammar for Java constructs and library invocations, as well as an algorithm that uses a customized natural language processing tool chain to extract information from free-form text queries. We present the results on a number of examples showing that our technique (1) often produces the expected code fragments, (2) tolerates much of the flexibility of natural language, and (3) can repair incorrect Java expressions that use, for example, the wrong syntax or missing arguments.

Categories and Subject Descriptors I.2.1 [*Artificial Intelligence*]: Applications and Expert Systems—Natural language interfaces; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis

General Terms Algorithms, Languages, Theory

Keywords Program Synthesis, Natural Language Processing, Autocompletion, Program Repair

1. Introduction

Application programming interfaces (APIs) are becoming more and more complex, presenting a bottleneck when solving simple tasks, especially for new developers. APIs contain many types and declarations, so it is difficult to know how to combine them to achieve a task of interest. Instead of focusing on more creative aspects of the development, a developer ends up spending a lot of time trying to understand informal documentation or adapting the API examples from developer forums or the documentation. Integrated development environments (IDEs) help in this task by listing declarations that belong to a given type but leave it to the developer to decide how to combine the declarations.

On the other hand, on-line repository host services such as GitHub [13], BitBucket [3], and SourceForge [30] are becoming increasingly popular, hosting a large number of freely accessible projects. Such repositories are an excellent source of code examples that the developers can use to learn API usage. Moreover, the large size and variety of code in these repositories suggests that they can be leveraged to create a more sophisticated IDE support. A natural first step is to perform code search [32], though this still leaves the user with the task of understanding context and adapting it to their needs. Several researchers have pursued the problem of generalizing from such examples in repositories, combining non-trivial program analysis and machine learning techniques [27].

In this paper, we present a new approach that synthesizes code appropriate for a given program point, guided by hints given in *free-form* text. We have implemented our approach in a system called *anyCode* and have found it to be useful in our experience (see Figure 9 for evaluation results and Figure 8 for examples used to configure *anyCode*). Our approach builds a model of the Java language, based on the corpus of code in repositories, and adapts the model to a given text input. In that sense, our approach combines some of the advantages of statistical programming language models [27], but also of natural language processing of input containing English phrases (which was previously done only for restricted APIs [21]). A prototype of our approach was presented in a short tool demonstration [14].

To construct type-correct expressions *anyCode* conceptually builds on our previous work, the InSynth tool [15, 16]. When using InSynth, a user indicates the desired result type of an expression; InSynth then generates ranked expressions of that type. In contrast, the input to *anyCode* has a much

[*] This work is supported in part by the European Research Council (ERC) project "Implicit Programming".

more flexible interpretation: it can represent any information related to the expression and can refer to any part of an expected expression. Furthermore, InSynth only uses the unigram language model [19, Chapter 4], which assigns a probability to a declaration based on its call frequency in a corpus. *anyCode*, in addition to unigram, uses the more sophisticated probabilistic context free grammar (PCFG) model [19, Chapter 14] to synthesize and rank the expressions.

To synthesize expressions, *anyCode* performs the following three key phases:

1. it uses natural language processing (NLP) tools [8, 24, 33] to structure input text and split it into chunks of words based on their relationships in the sentence parse tree;
2. it uses the structured text and unigram model to select a set of most likely API declarations, employing a set of scoring metrics and applying the Hungarian method [20] to solve the resulting assignment problem [5];
3. it uses the selected declarations, PCFG and unigram model to unfold the declaration arguments. The result is a list of ranked (complete or partial) expressions that *anyCode* offers to the developer, using the familiar code completion interface of Eclipse.

By introducing a textual input interface, we aim to automatically reduce the gap between natural languages and programming languages. *anyCode* allows the developer to formulate a query using a mixture of English and code fragments. *anyCode* takes into account English grammar when processing input text. To improve input flexibility and expressiveness we also consider word synonyms and other related words (hypernyms and hyponyms). We build a related word map based on WordNet [11], a large lexical database of English. We present a technique to make WordNet usable in our context by automatically projecting it onto the API jargon. We use these techniques along with the NLP tools to support the natural language aspect in *anyCode*. The techniques we implement in *anyCode* are inspired by stochastic machine translation. However, in contrast to machine translation, we had to overcome the lack of a parallel corpus relating English and Java, as well as the gap between an informal medium such as English and the rigorous syntax and type rules of a programming language such as Java.

One of our aims is to free the developer from the conventional rigid structure of a programming language when describing their intention. Our view is that IDE tools should allow a user to gloss over aspects such as the number and the order of arguments in method calls, or parenthesis usage, when these can be inferred automatically. The developers can then focus more on solving important higher-level software architecture and decomposition problems. Finally, we also hope to lower the entry for users learning to program—for whom syntax is often one of the first obstacles.

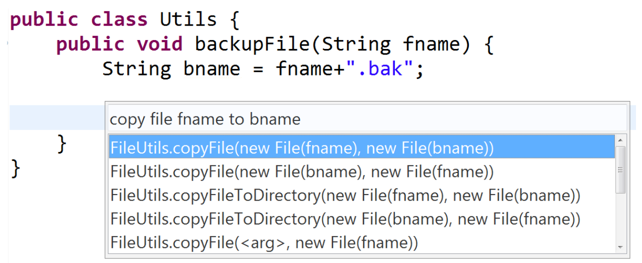
To achieve this, we find that a short text input that approximately describes the structure of the desired expression is the most convenient. To make the tool useful for programming, we also allow user's input to include literals and local variable names. Using such input, *anyCode* manages to synthesize valid Java code fragments. It can do that because it does not impose any strict requirement on the input: it has the ability to generate likely expressions according to the Java language model, and uses as much of the information from the input as it can extract to steer the expression generation toward the developer's intention.

Contributions To summarize, our paper makes the following contributions, spanning several individual techniques and areas:

- We present a unique pipeline of techniques that accepts text input and synthesizes a ranked list of (possibly partial) expressions. We combine customized natural language processing tools, a text-to-declaration matching algorithm, probabilistic context-free grammar (PCFG) models for Java applications, and an algorithm to generate expressions based on this information. The implementation of a tool, *anyCode*, including these techniques is publicly available [2].
- We describe a fast corpus analysis and extraction algorithm as well as its implementation, designed for building probabilistic Java language models. The algorithm extracts the occurrences of declaration compositions and declaration frequencies and builds PCFG and unigram models. We implemented and ran the algorithm on 1.8 million Java files to build a reusable probabilistic model for Java applications.
- We introduce an efficient approach for relating text to API declarations. Our approach prioritizes words based on their importance and position, both in the input text and inside declarations. To efficiently match text to declarations, we create appropriate indices, reduce matching to the assignment problem, and use the Hungarian method.
- We present a customized related-word map from each word to its related words. To build it, we use relations in WordNet and introduce a scoring technique that, for a given word, ranks and finds the closest related words. We use a set of API words to build the score and filter out irrelevant words.
- We introduce a benchmark set consisting of 90 pairs of free-form text input and Java expression output. Our benchmarks can be used to evaluate tools that map free-form text to Java expressions including API invocations.
- We present our experimental evaluation. We perform parameter tuning on 45 of the above examples and evaluate the system on the remaining 45. Our results suggest that our tool is helpful in practice. The evaluation also shows that the individual techniques we employed are important for obtaining such results.

2. Examples

We next illustrate the main functionality of *anyCode* through examples. The first example demonstrates *anyCode*'s interactive deployment, the text interface, and the use of program context to guide the synthesis.



```
public class Utils {
    public void backupFile(String fname) {
        String bname = fname+".bak";
          copy file fname to bname
    }     FileUtils.copyFile(new File(fname), new File(bname))
}         FileUtils.copyFile(new File(bname), new File(fname))
          FileUtils.copyFileToDirectory(new File(fname), new File(bname))
          FileUtils.copyFileToDirectory(new File(bname), new File(fname))
          FileUtils.copyFile(<arg>, new File(fname))
```

Figure 1. After the user inserts text input, *anyCode* suggests five highest-ranked well-typed expressions that it synthesized for this input.

2.1 Making a Backup of a File

Suppose that a user wishes to create a method that backs up the content of a file. The method should take the file name as a parameter and copy the content of the file to a new file with an appropriately modified name. To implement such a method, the user needs to identify the appropriate API, select the set of its declarations (typically method calls) and combine them into an expression. In practice, to perform this, an inexperienced user might follow these steps manually:

1. search the Internet or API documentation (if it exists) to find the examples of API use
2. select the most suitable example
3. copy-paste it into the working editor with code, and
4. edit the example to adapt it to the current program context, using appropriate values in the program scope.

anyCode offers an *automated* approach as an *alternative* to the above manual steps. Suppose that a user writes an incomplete piece of code of the method that takes the parameter fname storing the file name, as shown in Figure 1, and also introduces a local variable bname storing the name of the backup file. When the user invokes *anyCode*, a pop-up text field appears where she can insert the text. Assume she enters the text "copy file fname to bname", specifying her desire to copy the file content. *anyCode* automatically extracts the program context from Eclipse and identifies words fname and bname in the input as values referring to a parameter and a local variable. *anyCode* then uses this information to generate and present several ranked expressions to the user. When the user makes her choice, the tool inserts the chosen expression at the invocation point. In this example, *anyCode* works for less than 70 milliseconds (which is typical) and then presents five solutions of which the first one copies the file fname content to a file with name bname:

FileUtils.copyFile(**new** File(fname), **new** File(bname))

This is a valid solution; it uses the method FileUtils.copyFile from the popular "Commons IO" library.

2.2 Obtaining Screen Refresh Rate

We next consider a scenario where a user wishes to determine a screen's display refresh rate. The user can simply invoke *anyCode* with the free-form query "get display refresh rate". In response, *anyCode* synthesizes and suggests the following expressions:

1 GraphicsEnvironment.getLocalGraphicsEnvironment()
    .getDefaultScreenDevice().getDisplayMode()
    .getRefreshRate()
2 Calendar.getInstance()
3 DisplayMode.REFRESH_RATE_UNKNOWN
4 Policy.getPolicy().refresh()
5 System.getProperty(''?'')

The first suggestion turns out to be the desired one. Note that this suggestion is not the shortest possible one, and it includes declarations that are not computed from the user's input in any straightforward way. Whereas methods getRefreshRate and getDisplayMode all contain input words (get, refresh, rate and display), the methods getDefaultScreenDevice and getLocalGraphicsEnvironment do not. To determine the need to invoke getLocalGraphicsEnvironment in the presence of getRefreshRate, we use probabilistic language model for Java and its API calls, derived from a corpus of code. The model allows us to favor more popular declarations *and* declaration compositions, so it can determine that the additional declarations are a common way to build expressions containing the more obviously relevant declarations.

2.3 Creating a Polygon

Suppose now that a developer wishes to obtain an instance of the class Polygon. Let us say she already defined polygon points using two equal-length integer arrays x and y, for x and y coordinates, respectively. If the developer inserts the following free-form query:

polygon x y

anyCode generates the following suggestions:

1 **new** Polygon()
2 **new** Polygon(x, y, 0)
3 **new** Polygon(y, x, 0)
4 **new** Polygon(x, y, y.length)
5 **new** Polygon(x, y, x.length)

This example shows the importance of presenting multiple solutions to the user, and further illustrates some of the preference criteria built into *anyCode*. Note that suggestions 4 and 5 are both acceptable. The solutions 2 and 3 use integer literal as a third argument. The composition of the Polygon constructor with the integer literal as the third argument is more popular than the composition of the same constructor and the length field that appears in the last two expressions. For this reason, solutions 2 and 3 a have higher score than 4 and 5 (constant 0 as the third argument is merely a placeholder, expected to be adapted by the developer). On the other hand, solution 1 has the best score because the Polygon constructor without parameters is more popular than the constructor with three parameters. Furthermore, solution 2 has a better score than 3 and solution 4 has a better score than 5. The reason is that 2 and 4 follow the order of local variables x and y in the input query, where x comes before y. *anyCode* uses this scoring technique to differentiate suggestions that would otherwise have the same score.

2.4 Creating a New File

In the next example we demonstrate the use of semantically related words. Suppose that the user wishes to discover how to generate a new file on disk. She may use a query such as "make file". In response, *anyCode* generates the output:

```
1 new File(⟨arg⟩).createNewFile()
2 new File(⟨arg⟩).isFile()
3 new File(⟨arg⟩)
4 new FileInputStream(⟨arg⟩)
5 new FileOutputStream(⟨arg⟩)
```

Note that the word make does not appear among the solutions, simply because API designers chose to use the related word create instead. *anyCode* succeeds in finding the solution because it considers not only the words literally appearing in the input, but also words *related* to them, which, in this case, includes create. *anyCode* uses a custom related word map to compute the relevant words. We built this map by automatically processing and adapting WordNet, a large lexical semantic network of English words.

Additionally, all suggestions in this example represent templates that include the symbol ⟨arg⟩ that marks the places where local variables are often used. The main reason why we present templates is that a user often enters incomplete input, for which the best solution is an incomplete output, i.e., a template. Had we insisted only on completed expressions, we would miss many interesting solutions that are more convenient for such an incomplete input. For this reason, *anyCode* treats a textual input as potentially incomplete and tries to find complete as well as incomplete solutions.

2.5 Checking File Permission

Consider the following input of a developer:

```
check file "text.txt" "read" permission
```

This example shows that the input may contain string literals in quotation marks, which the tool may reuse in the generated code fragments (*anyCode* also allows a user to directly insert number and boolean literals). For this free-form input *anyCode* outputs:

```
1 AccessController.checkPermission(
          new FilePermission("text.txt", "read"))
2 AccessController.checkPermission(
          new FilePermission("read", "text.txt"))
3 new File("text.txt")
4 new FileInputStream("text.txt")
5 new FileInputStream("read")
```

The constructors File and FileInputStream are often used and more popular than other declarations that appear among the solutions. However, the expressions under 1 and 2 are ranked higher than others because they have a higher usage of the input text elements. For instance, solution 3 does not refer to one of the string literals in the input, nor does it include words check and permission. The synthesis algorithm and our scoring techniques favor solutions with the greater input coverage. In this example, the first expression performs the desired task.

2.6 Reading from a File

In our final example we show that our input interface may also accept an approximate Java-like expression. For instance, if a user attempts to write an expression that reads the file, in the first iteration she may write the expression:

```
readFile("text.txt","UTF−8")
```

Unfortunately, this expression is not well-typed according to common Java APIs. Nevertheless, if *anyCode* takes such a broken expression, it pulls it apart and recomposes it into a correct one, suggesting the following solutions:

```
1 FileUtils.readFileToString(new File("text.txt"))
2 FileUtils.readFileToString(new File("UTF−8"))
3 FileUtils.readFileToString(⟨arg⟩)
4 FileUtils.readFileToString(new File(⟨arg⟩))
5 FileUtils.readFileToString(new File("text.txt"),"UTF−8")
```

anyCode first transforms the input by ignoring the language specific symbols (e.g., parenthesis and commas). It then slices complex identifiers, so called *k-words*, into single words. Here, readFile is a 2-word that gets sliced into read and file. Despite the loss of some structure in treating the input, our language model gives us the power to recover meaningful expressions from such information. This shows how *anyCode* can be used as a simple expression repair system. The desired solution is ranked fifth because it uses a version of readFileToString method with two arguments, which appears less frequently in the corpus than the simpler versions of the method. The repair functionality is not a pre-engineered feature of our system, but rather a property that emerges as a consequence of a robust interface for handling text input.

We have evaluated our system on a number of examples. Figure 9 shows 45 text queries and the code that we expected to obtain in return. The "All" column indicates the rank on

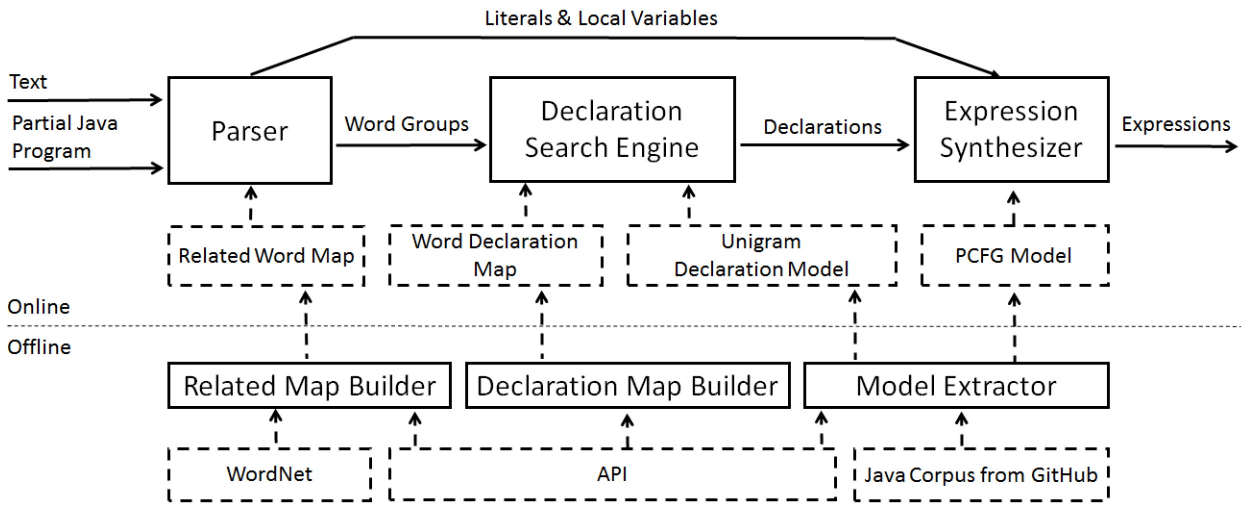


Figure 2. *anyCode* system overview. *Offline system* components run only once at tool setup time to produce the information for the online system components. *Online system* components run continuously as a part of the Eclipse plugin.

which the expression was found with all features of our system turned on, as discussed in Section 11. In addition to these evaluation examples, we present another 45 examples, in Figure 8, used to tune *anyCode*. For these examples *anyCode* also successfully synthesizes expected expressions.

3. System Overview

In this section we give a high-level description of the main components of our system. At the top level, we divide the system into the online and the offline part (see Figure 2).

3.1 Online System

The online part of *anyCode* interactively suggests expressions to a developer within the Eclipse IDE. Input to the online part consists of: 1) a textual description, explicitly entered by the developer, and 2) a partial Java program with a position of the cursor, which *anyCode* extracts automatically from Eclipse. *anyCode* uses the input to generate, rank, and present expressions to the developer. The key components of *anyCode* are: the input parser, the declaration search engine, and the expression synthesizer. The method getExpressions is the main method that performs these steps, as outlined in Figure 3.

Parser The first goal of parsing is to identify structure of the input text using a set of natural language processing tools. *anyCode* uses the structure to group input words into WordGroups. We group the words because we expect a user to insert the text that corresponds to several declarations. Grouping according to the rules of English helps the system identify these declarations from multiple input words. Next, to make the input more flexible, *anyCode* completes the words given in the input with semantically related words (Section 7). Finally, to complement natural language input and to be able to accept broken Java-like expressions, *anyCode* uses program context, a set of all local variables visible from the cursor point, to mark local variables in the input text. Moreover, *anyCode* also identifies literals in the input text. The parsing phase is implemented as the parse method; Section 4 describes it in more depth.

```
getExpressions(text, partialProgram, N, M, S):
    // Parsing
    context ← extract(partialProgram)
    (WordGroups, Literals, Locals) ← parse(text, context)
    // Declaration Search
    DeclGroups ← declSearch(WordGroups,API,Unigram,M)
    // Expression Synthesis
    ExPCFG ← extend(PCFG, Literals, Locals)
    return synth(DeclGroups, ExPCFG , S, N)
```

Figure 3. A high level description of the online system.

Declaration Search Engine In the declaration search engine, *anyCode* uses WordGroups to find a subset of API declarations that are most likely to form the final expressions. *anyCode* tries to match WordGroups against declarations in our API collection. To perform matching, *anyCode* extracts a list of words from declarations and matches them against the words in WordGroups. *anyCode* estimates the declaration matching score based on the number of words that it matched. We use the matching score and declaration Unigram [19, Chapter 4] score (representing the declaration popularity or the frequency in the corpus) to calculate total declaration score. *anyCode* then uses the score to select top M declarations for each $w \in$ WordGroups and to form declaration groups DeclGroups. In summary, the method declSearch transforms each word group into a declaration group. This approach accounts for the ambiguity in the input text while mapping the input to expected declarations (see Section 8 for further details).

Expression Synthesizer In the last phase *anyCode* uses declaration groups, DeclGroups, and a probabilistic context free grammar (PCFG) model [19, Chapter 14] to synthesize expression. Our PCFG model contains the statistics on declaration compositions pre-collected from a large Java source code corpus hosted on GitHub. The compositions are encoded as production rules with appropriate probabilities. The model determines how *anyCode* should compose declarations to obtain expressions that are likely to occur in real-world applications. To include literals and local variables we extend PCFG in the method extend with the production rules for literals and local variables, obtaining the extended model ExPCFG. Next, for each declaration in DeclGroups, the method synth tries to unfold declaration arguments following the ExPCFG model in S steps. Given a declaration, ExPCFG suggests the most likely declarations that fill in the argument places. Additionally, the method synth assigns scores to synthesized expressions based on ExPCFG and declaration scores. Finally, synth sorts all the expressions based on the score and outputs the first N to the user. We describe this phase in Section 6.

3.2 Offline System

Unlike the online system that runs interactively inside Eclipse IDE, the offline system contains components that we run once before a user starts interacting with *anyCode*. We use the offline components to build necessary data structures for *anyCode* to operate efficiently and effectively. These include: *Word Declaration Map*, which optimizes matching between input words and declarations, *Related Word Map*, which maps words to related API words, and, finally, PCFG and Unigram models.

Declaration Map Builder To efficiently match WordGroups with API declarations in the declaration selection phase we build the Word Declaration Map. We use a pre-collected set of API declarations to create a map from each word to the set of declarations that contain that word. In the first step we collect API declarations from popular APIs and packages. Next, we use the set of natural language processing tools to extract words from each declaration. The words belong to names as well as argument and return types of the declarations. Finally, we use the words and declarations that contain the words to build the Word Declaration Map.

Related Map Builder To make the input more flexible we complement the input text with related words. For this purpose, we build the Related Word Map that represents the map of related words computed from WordNet [11], a large lexical database of English. WordNet groups synonyms into sets and defines other relations between those sets. We use WordNet relations and our API collection to build Related Word Map that maps words to related API words.

Model Extractor Model Extractor extracts PCFG and Unigram models from the Java source corpus hosted on GitHub. We built a lightweight compiler front-end for Java to compile the files and the extractor to extract the model from the compiled units. First, we use the compiler to build a symbol table, in order to perform type checking and to build approximate ASTs. The symbol table contains the information on local symbols as well the information on API declarations. Approximate ASTs, unlike complete ASTs, might miss type and symbol information in some AST nodes. Second, we extract as precisely as possible the statistics from the approximate ASTs, obtaining PCFG and Unigram models.

4. Parsing

During the parsing process our system resolves some of the ambiguity of the input text and identifies its deeper structure. The parsing helps us group the words from the input text and also allows us to extract and group the words from the declaration signatures (e.g., declaration names, argument and return types). More specifically, we use parsing to structure the input text in the method parse (Figure 3) as well as to process the API declaration in Declaration Map Builder.

In the next phase (Section 5), we use the input word groups to match the declaration words and thus select the most relevant API declarations. Parsing is an important step that allows efficient declaration selection. It helps *anyCode* to effectively use the input text and select only a few relevant declarations from our API collection (with over 10,000 declarations), significantly reducing search space.

In the sequel, a k-word denotes a chain of k English words connected without a whitespace or the underscore between them as often used in Java identifiers. The words are separated at places where the underscore appears or where a small letter meets a capital letter. Often, declaration names are k-words (e.g., "readFile" is a 2-word that contains the words "read" and "file"). A 1-word is a single English word. We say that a token is either a k-word, a literal or a local variable name. Among literals, *anyCode* supports numbers, strings, and booleans.

4.1 Input Text Parsing

To describe the parsing process, we use two examples, showing different phases of the parse method in Figure 3.

In the first example, a user inserts 'copy file fname to "C:/user/text2.txt"', as shown in the row zero of Figure 4 (fname is the local variable). We represent each phase of the parsing by one of the rows 1-5 and 7. Each row contains a name of a phase and an output. In the first phase we remove Java symbols (e.g., commas and parenthesis) and whitespace characters. In the second phase, we identify tokens and mark literals as well as local variables with labels "Lit" and "Var", respectively. In our example, we identify one local variable fname and one literal "C:/user/text2.txt". In the third phase, row 3, we decompose k-words into single words. In Figure 4,

	Phase Name	Word Type	Output				
0	Text Input		copy file fname to “C:/user/text2.txt”				
1	Removing Java Symbols		copy	file	fname	to	“C:/user/text2.txt”
2	Tokenization		copy	file	Var(fname)	to	Lit(“C:/user/text2.txt”)
3	Decomposing k-words		copy	file	Var(fname)	to	Lit(“C:/user/text2.txt”)
4	Swap Literals & Locals		copy	file	string	to	string
5	POS Tagging		copy/Verb	file/Noun	string/Noun	to/To	string/Noun
7	Grouping		Group 1	Group 2	No Group	Group 3	No Group
		Primary	copy/Verb	file/Noun		to/To	
		Secondary	file/Noun, to/To	string/Noun		string/Noun	

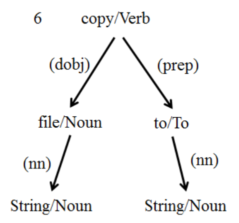


Figure 4. Phases of parsing the “copy file” example.

	Phase Name	Word Type	Output			
0	Text Input		readFile(“text.txt”,“UTF-8”)			
1	Removing Java Symbols		readFile		“text.txt”	“UTF-8”
2	Tokenization		readFile		Lit(“text.txt”)	Lit(“UTF-8”)
3	Decomposing k-words		read	file	Lit(“text.txt”)	Lit(“UTF-8”)
4	Swap Literals & Locals		read	file	string	string
5	POS Tagging		read/Verb	file/Noun	string/Noun	string/Noun
7	Grouping		Group 1		No Group	No Group
			Subgroup 1	Subgroup 2		
		Primary	read/Verb	file/Noun		
		Secondary		string/Noun, string/Noun		

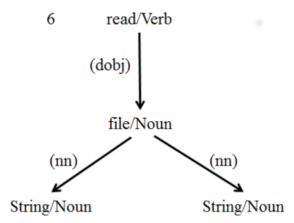


Figure 5. Phases of parsing the “read file” example.

all words are already single words, thus the output remains the same as in the previous phase.

Because we expect local variables and literals to appear as arguments in an expression, we extract their types and use them to search for the declarations with the same argument types. Therefore, in the next phase, row 4, we replace literals and local variables with their types. Both fname and “C:/user/text2.txt” are replaced by string.

In general, mapping the input words to the declarations is ambiguous. For instance, each word can be mapped to a distinct declaration (e.g., copy to the declaration with the name copy, file to the declaration with the name file, etc.) or a group of words can be mapped to the same declaration (e.g., copy and file can be mapped to the method with the name copyFile, and file and string can be mapped to the constructor **new** File(String)). Therefore, to resolve this ambiguity we need to know what is the most likely word grouping so that the words in the groups can be mapped to the most relevant declarations. For this reason, we employ NLP tools to obtain deeper structure in text and use it to effectively group words.

We thus continue with the fifth phase, row 5, where we use the Stanford CoreNLP [8, 24, 33] tools to lemmatize words, put them into their canonical form (e.g., “good” is the lemma of “better”), and tag them with Part-of-Speech (POS) tags. The POS tags are assigned based on the lexical content of a word and its position in the sentence (the tags can be Verb, Noun, Adjective, Adverb, etc.). We perform this step for two reasons: (1) we observe that verbs mostly appear in the API method names, where non-verb words appear almost anywhere (in argument types as well in method and constructor names), and (2) a POS-tagged sentence, a list of POS-tagged words, is the input to the NL parser. In phase 6, we pass the tagged sentence to the Stanford NL parser and obtain a semantic graph, shown on the right-hand side of Figure 4 (in general a semantic graph is a directed acyclic graph, but in our examples are a tree). The nodes are words and the edges are relations. For instance, the edge that goes from copy to file, denoted with “dobj”, says that file is a direct object of copy. In the final phase, row 7, we use the graph to group the words. For each word w, except for the literal and local variable types (two “string” words), we form a group. The group contains w and its children words. The children words are the direct children of w at the opposite sides of its outgoing edges. For instance, the first group contains words copy as w, and file and to as w’s children.

We expect that some words in the group may match with declaration names and others with types. For instance, the second group contains the word file that may match the name of the constructor **new** File(String) (the **new** File part) and the word string that may match the constructor’s argument type (the String part in the parenthesis). For this reason, we make a distinction between the words in the word group. Our primary alignment strategy is to match w with the words from the declaration names. By the same strategy, we expect w’s children to match with the words from the declaration types. Although this is the primary strategy, it is not the only one. For instance, an alternative strategy is that the children words may also match the words from the declaration names. This strategy is useful when we want the first group words copy as w and file as w’s child, to match the name of the copyFile declaration. We also referred to w as the primary word. We call its children *secondary* words.

We expect that the user will more often insert declaration name words than the type argument words. For instance, if a user types a word file it is more likely that she refers to the declaration that contains file in the name than to the declaration that contains file only among the arguments. Although we consider all declarations during the matching phase, we give priority to the declarations that have file in the name over the declarations that have file only among the argument types. One way to implement this is to give higher priority to primary than to secondary words. In Section 8 we explain in more detail our scoring and alignment model with different strategies using primary and secondary words.

In our first example we have three different word groups denoted with numbers 1-3. The group i contains all the words below the number i. Therefore, Group 1 contains the primary word copy and the secondary words file and to. We do not create groups for the "string" literals because they are expected to match argument types.

In the case of a k-word we also want to take into account the suggested connection among the words that appear in the k-word. In the second example, Figure 5, the most interesting is phase 3, where we split the 2-word readFile into file and read, and phase 7, where we form the groups based on k-words. In the case of k-words, where $k > 1$, we create a single group for the entire k-word to take the connection into account. However, for every single word in k-words we create a subgroup. For instance, Group 1 contains subgroups Subgroup 1, with the primary word read, and Subgroup 2, with the primary word file and its secondary words based on the semantic graph (Figure 5). Subgroup 1 does not contain any secondary words because the word file is already the primary word of Subgroup 2.

To satisfy the constraint that verbs often appear in the declaration names, we omit verbs from secondary words. Additionally, another important constraint that we consider is that each group contains at least one non-verb word because we observe that declaration signatures usually contain at least one non-verb word.

4.2 Declaration Parsing

Previously, we demonstrated how we parse the text input and obtain word groups. We use the word groups to select the declarations. The natural way to do this is to match the words from the groups against the words that appear in the declarations (including the words that appear in the declaration name, argument and return types). To obtain declaration words we also use parsing in the pre-processing stage. We extract the words from the declaration name and types. We parse each declaration using the same parsing technique, phases 1-5, we described before. However, we do not make different group words, but only put declaration name words into the set of primary declaration words, and type words into the set of secondary declaration words. For instance, the declaration copyFile(File, File):Unit will have the primary group that contains copy and file and the secondary group that contains words file two times and unit.

Our goal will be to maximize the matching score between the input group words and the declaration words. Intuitively, the score will be higher if the primary words from the input group and the primary words from the declaration match, and if the secondary words from the input group and the

Word Group 1	
copy/Verb	
file/Noun, to/To	

Declaration Group 1	
Name	Type
copyFile	(File, File):Unit
copyFileToDirectory	(File,File):Unit
copyURLToFile	(URL, File):Unit
copyOf	(X[], int):X[]
copyFile	(File, File, boolean):Unit

Figure 6. The declaration selection for "copy file".

secondary words from the declaration match. Any deviation from this will have a smaller matching score. In Section 8 we show the framework to calculate the matching score.

5. Declaration Selection

The previous section outlined how our system groups input words using NLP techniques to obtain Word Groups. In the next phase we use Word Groups to select the set of the most relevant declarations.

The declarative description of the selection algorithm is simple. For each word group in Word Groups we: (1) match the word group with all declarations (words from the word group are matched against the words in declarations), (2) calculate the matching score, (3) use the matching score and unigram score to calculate the declaration score (in Section 8.1 we give more details about scoring), and (4) select the M declarations with the top declaration score.

The challenge is to efficiently match the words from a word group with the words in all API declarations. Matching the entire collection of API declarations with a word group is impractical because the collection contains over 10,000 declarations and matching score calculation is an expensive operation. To reduce the number of calculations we built in advance the Word Declaration Map, briefly mentioned in Section 3.2, which maps a word w to the set of API declarations D that contain w. In practice, D is much smaller than the API collection. We use the parsing techniques previously presented to identify and extract words from the declarations and to build the map. In Figure 6 we present the result of the selection using Groups 1 from Figure 4. The detailed description of the selection algorithm is given in Figure 10.

6. Expression Synthesis using PCFG model

In this section we describe the algorithm that synthesizes the expressions using the PCFG model. Because the algorithm relies on PCFG, we first explain the model and later the expression synthesis.

6.1 Probabilistic Context Free Grammar Model

We use PCFG to guide the synthesis algorithm and to rank the expressions. Our PCFG model consists of two parts: (1) the incomplete PCFG extracted from the source corpus, and (2) the extension with the PCFG production rules for local variables and literals which appear in the user's local context and input.

6.1.1 Incomplete Model from Corpus

We use PCFG model to encode API declaration compositions from the source corpus. At the invocation site, the declaration f may simultaneously compose many declarations that appear in the argument places of f. Consider the expression f(a,b(c, e)). Here f composes simultaneously with a and b, where b simultaneously composes with c and e. We referred to the simultaneous compositions f(a,b) and b(c,e) as *multi-compositions*.

Let SType be the set of all simple ground API types. The productions of our PCFG, that encode multi-compositions, are described by the following grammar:

$$\begin{aligned}&\mathsf{decl\_prod} ::= \mathsf{?Decl(d)} \xrightarrow{p} [\mathsf{hole.}]\ \mathsf{d}\ [(\mathsf{hole},\ ...,\ \mathsf{hole})]\\ &\mathsf{hole} ::= \mathsf{?Decl(d)} \mid \mathsf{?Var(type)} \mid \mathsf{?Lit(type)}\\ &\mathsf{type} \in \mathsf{STypes} \qquad \mathsf{d} \in \mathsf{Decls}\end{aligned}$$

We first introduce the notion of *holes*. We consider three kinds of holes: a declaration hole, ?Decl(d), a local variable hole, ?Var(type), and a literal hole, ?Lit(type). Each hole is a nonterminal, which contains additional information at the same time. A declaration hole stores declaration d, whereas a local variable and a literal hole store type.

A declaration production rule, decl_prod, encodes how a declaration d is simultaneously composed with holes. Intuitively, each declaration production rule encodes one multi-composition. For instance, f(a,b) is encoded as $\mathsf{?Decl(f)} \xrightarrow{p}$ f(?Decl(a), ?Decl(b)). First, note that declaration d appears at both sides of the rule. On the left-hand side, it appears as part of the hole and nonterminal ?Decl(d). On the right-hand side, the same declaration d appears as terminal and can be preceded by a hole which represents a receiver and can be followed by argument holes. An argument hole can be a local variable hole or a literal hole. In the PCFG model that we extract from the source corpus, we do not keep rules for local variable holes and literal holes. Finally, the probability p says how frequently multi-composition occurs in the corpus.

6.1.2 Model Extension from Local Context and Input

We complete the incomplete corpus model with rules for variables and literals that belong to the users local context and that appear in the input:

$$\begin{aligned}&\mathsf{e\_prod} ::= \mathsf{?Var(type)} \xrightarrow{p} \mathsf{var} \mid \mathsf{?Lit(type)} \xrightarrow{p} \mathsf{lit}\\ &\mathsf{var} \in \mathsf{Vars} \qquad \mathsf{lit} \in \mathsf{Literals}\end{aligned}$$

This way, we use the extension to allow local variables and literals to appear in the synthesized expressions. Finally, we include rules for the missing local variables and the default values of the literals:

$$\begin{aligned}&\mathsf{def\_prod} ::= \mathsf{?Var(type)} \xrightarrow{1} \langle \mathsf{arg} \rangle \mid \mathsf{?Lit(type)} \xrightarrow{p} \mathsf{def\_literal}\\ &\mathsf{def\_literal} ::= \text{"?"} \mid 0 \mid \mathsf{false}\end{aligned}$$

The symbols ⟨arg⟩, ?, 0 and false are default termination symbols. We include these rules to allow a developer to modify the expressions after the synthesis.

6.2 Expression Synthesis

The goal of the synthesis algorithm is to use declarations from the declaration groups to synthesize relevant expressions. The algorithm should favor the expressions that are more likely to appear in the real-world projects. For this reason, we calculate expression scores which we explain in more detail in Section 9.

We can describe the synthesis algorithm as follows. For each declaration group DG_i and for each declaration $\mathsf{d} \in \mathsf{DG}_i$, we create hole ?Decl(d). Then, we use PCFG to unfold ?Decl(d). To unfold hole ?Decl(d), we use the production rules from PCFG where d appears on the left-hand side, i.e., all rules of the form $\mathsf{?Decl(d)} \xrightarrow{p} [\mathsf{hole.}]\ \mathsf{d}\ [(\mathsf{hole},\ ...,\ \mathsf{hole})]$. Then we substitute the hole with the right-hand side [hole.] d [(hole, ..., hole)]. While unfolding holes, we create *partial expressions*. Unlike complete expression, a partial expression contains holes. We continue unfolding holes in the partial expressions, creating new partial expressions. We repeat this process some limited number of steps S. This way, for each DG_i, we create a partial expression group PEG_i which contains partial expressions synthesized using PCFG starting from the declarations in DG_i. If we encounter a local variable hole or a literal hole we search for rules which have ?Var(type) and ?Lit(type) on the left-hand side, respectively.

The number of partial expressions in each new step can grow exponentially and for this reason we apply a heuristic search. To make the synthesis more efficient we use the *beam* search algorithm. In every step, the algorithm takes all the partial expressions PE from the previous step, and unfolds a single hole per partial expression $\mathsf{pe} \in \mathsf{PE}$. This creates new partial expressions and only N with the best score are passed on to the next step. As mentioned, we repeat this process S times. The algorithm synthesizes expressions using beam search that does not guarantee an optimal solution. Therefore, to make the algorithm more effective, we use an approximation that synthesizes more expressions than the N expressions that will be shown to a user. The system sorts the larger set using the expression scores and outputs the best N expressions to the user.

To build more interesting partial expressions that compose partial expressions from different groups PEG_i and PEG_j, we build the mechanism that first detects a hole ?Decl(d) in expression $\mathsf{pe}_i \in \mathsf{PEG}_i$, that can be substituted with expression $\mathsf{pe}_j \in \mathsf{PEG}_j$. The hole ?Decl(d) can be substituted with pe_j if d is the topmost declaration of the pe_j (e.g., f is the topmost declaration of f(a,b(c,e)) expression). The mechanism first replaces ?Decl(d) with *connector* $\mathsf{c(d,\ PEG}_j)$ that keeps enough information to be later substituted with pe_j. We use connectors because at the time we create $\mathsf{c(d,\ PEG}_j)$, the expression pe_j might not yet exist, but we have enough information to efficiently obtain it in the future once it is constructed. This allows us to have a lot of flexibility in which portions of the search space we explore first. This is important for two reasons: (1) we can use the scoring technique (Section 9) to navigate the search and synthesis towards the most interesting parts of the search space, and (2) we can efficiently parallelize the synthesis by constructing separately each PEG_i in parallel. In the final phase, our mechanism substitutes the connectors with concrete expressions to obtain complete expressions. Further details on the synthesis algorithm, partial expressions, and connectors are shown in Appendix A.2.

7. Related Word Map: Modifying WordNet

To support inputs that do not strictly follow the actual words of API declarations, we use WordNet [11], a large lexical database of English words. WordNet groups words into synsets, which are sets of synonyms. It also keeps different relations between synsets which include antonyms, hypernyms, and hyponyms. A single word can belong to many different synsets. Each synset represents a different meaning of the word. Moreover, WordNet keeps explicit textual description associated with every synset. The description is an English text which describes the meaning of the synsets. We use them to discover synsets that are close to the "programming/API" application. On the other side, our API declaration collection contains the words that are only a subset of English. We referred to them as *API words*.

We decide to enrich each word group (Section 4) with related words of primary words. To find the related words efficiently, we build Related Word Map that maps primary words to the API words using WordNet synsets and relations. More precisely, it maps the primary word w to the set of words R with associated scores. The score says how related w and $\mathsf{w}_r \in \mathsf{R}$ are. For instance, the table below shows the words related to the verb "make".

Word	create	press	chop	yield	cut	clear
Score_r	1.0	1.0	1.0	0.86	0.86	0.85

Based on the related score score_r, "create" is more related to "make" than "clear". We build the related score score_r using the relation type and textual descriptions associated with the synsets. There are two cases. In the first case, let w_{sh} belong to the same synset S as the primary word w. Then w_{sh} has score_{rs} equal to the percentage of the API words that appear in the description of S (first we remove frequent words from the descriptions and later we calculate the score). This way, we favor synsets whose descriptions have more API and programming jargons. In the second case, let S be in hypernym or hyponym relation with the synset S_h and let the primary word w be in S and the word $\mathsf{w}_{sh} \in \mathsf{S}_h$. Next, let the score of S be again score_{rs} and the score of S_h be score_{rh}.

Then the related score of w_{sh} with respect to w is the product of score_{rs} and score_{rh}. To summarize:

$$\mathsf{score}_r(\mathsf{w}_{sh}) = \begin{cases} \mathsf{score}_{rs}, & \mathsf{w}_{sh} \text{ synonym of } \mathsf{w} \\ \mathsf{score}_{rs} \cdot \mathsf{score}_{rh}, & \mathsf{w}_{sh} \text{ (hype/hypo)nym of } \mathsf{w} \end{cases}$$

8. Declaration Score

We would like to select declarations that are both well matched with the input words and popular in the corpus. Therefore, we compose the declaration score score_d using the matching score score_m and the declaration unigram (popularity) score score_u in the following way:

$$\mathsf{score}_d = \mathsf{c}_m \cdot \mathsf{score}_m + \mathsf{c}_u \cdot \mathsf{score}_u$$

In the rest of the section we explain how we build the scores score_m and score_u. Later, in Section 11.1, we explain how we learn values for the coefficients c_m and c_u shown in Figure 12 of the Appendix.

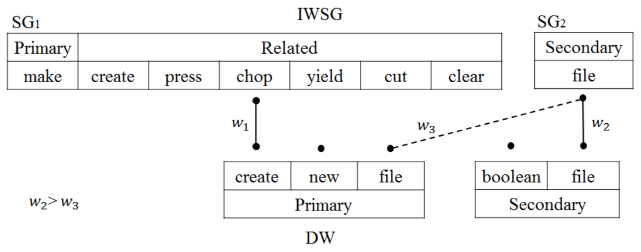


Figure 7. The example of matching the input group with declaration createNewFile.

8.1 Matching Score

The matching score is the measure of the matching quality between an input word group and a declaration. In Section 4 we showed that a word group has primary and secondary words. We also mentioned that a declaration has primary and secondary words. The primary belongs to the declaration name and the secondary belongs to types of the declaration.

The matching can be represented as the matching in the bigraph with the two disjoint sets DW and IWSG, as shown in Figure 7 (note that we also use POS tags, but for simplicity we omit them in the examples and this section). DW is the set of declaration words, in the example this includes: 1) primary words create, **new** and file, and 2) secondary words boolean and file. IWSG contains the words from the input word group. It groups them into subgroups. Each subgroup contains a secondary word or a primary word with its related words. This way, we include related words used to substitute primary words when the substitution will result in greater matching score. In the example, we have two subgroups. One represented with the primary word make and its related words and the other represented with the secondary word file.

Now, we describe the matching. A subgroup $\mathsf{SG} \in \mathsf{IWSG}$ can be matched with word $\mathsf{w} \in \mathsf{DW}$ if there is word $\mathsf{w}_{sg} \in$

SG with the same lexical content like w. For instance, a secondary word file from SG_2 has the same lexical content as the primary and secondary words file in DW. Therefore, SG_2 can be matched with any of the two words from DW. However, the matching quality between the two possible matchings might be different. Therefore, for each possible matching pair (SG, w), $\mathsf{w} \in \mathsf{DW}$, we first calculate the *matching weight*. Then, our matching problem is reduced to finding a maximum weight matching in a weighted bipartite graph. This is a well-known assignment problem [5] and to solve it, we use Hungarian method [20]. Then, our matching score score_m is the maximum matching score that we obtain. For instance, as shown in Figure 7, the maximum matching is represented with solid lines.

Now it remains to explain how we calculate matching weight. We define the matching weight weight_m between SG and $\mathsf{w} \in \mathsf{DW}$ as the maximum of matching scores between all words from SG and w, i.e.:

$$\mathsf{weight}_m(\mathsf{SG},\ \mathsf{w}) = \max(\mathsf{weight}_m(\mathsf{w}_{sg},\ \mathsf{w})), \quad \mathsf{w}_{sg} \in \mathsf{SG}$$

Between $\mathsf{w}_{sg} \in \mathsf{SG}$ and $\mathsf{w} \in \mathsf{DW}$, the matching weight weight_m is defined as the product of the word importance weights and the word matching type weight:

$$\mathsf{weight}_m(\mathsf{w}_{sg},\mathsf{w}) = \mathsf{weight}_i(\mathsf{w}_{sg})\cdot\mathsf{weight}_i(\mathsf{w})\cdot\mathsf{weight}_t(\mathsf{w}_{sg},\mathsf{w})$$

Word Importance Weight We expect a user to most likely insert words that denote declaration names. For this reason, we favor primary over secondary and related words. Thus, the importance weight weight_i is defined as follows:

$$\mathsf{weight}_i(\mathsf{w}) = \begin{cases} \mathsf{c}_p & \text{w is primary word} \\ \mathsf{c}_s & \text{w is secondary word} \\ \mathsf{c}_p \cdot \mathsf{c}_r \cdot \mathsf{score}_r(\mathsf{w}) & \text{w is related word} \end{cases}$$

Note that we learn coefficients c_p, c_s and c_r under the constraint $\mathsf{c}_p > \mathsf{c}_s$ using methods in Section 11.1.

Word Matching Type Between the words in IWSG and the words in DW there are four possible combinations of matching that $\mathsf{weight}_t(\mathsf{w}_{sg},\ \mathsf{w})$ captures:

$$\mathsf{weight}_t(\mathsf{w}_{sg},\mathsf{w}) = \\ = \begin{cases} \mathsf{c}_{pp} & \text{both words are primary} \\ \mathsf{c}_{ps} & \mathsf{w}_{sg} \text{ is primary and w is secondary word} \\ \mathsf{c}_{sp} & \mathsf{w}_{sg} \text{ is secondary and w is primary word} \\ \mathsf{c}_{ss} & \text{both words are secondary} \end{cases}$$

Again, we learn coefficients c_{pp}, c_{ps}, c_{sp}, and c_{ss} using methods in Section 11.1. While tuning *anyCode* we would also like to satisfy the constraint $\mathsf{c}_{pp} > \mathsf{c}_{ss} > \mathsf{c}_{sp} > \mathsf{c}_{ps}$(in our example, due to $\mathsf{c}_{ss} > \mathsf{c}_{sp}$, w_2 is greater than w_3). This constraint encodes our intuition that the primary-primary matching is the most important and the primary-secondary is the least important. The latter is the least important because we do not want to use the primary input words to select the declarations by matching them with secondary words. In contrast, we expect that secondary input words might appear often in the names of declarations and for this reason they might often match with primary declaration words.

8.2 Unigram Score

The unigram model [19, Chapter 4] assigns a probability to each declaration based on the corpus occurrence frequency. The higher the declaration frequency, the higher the probability. We smooth the model by assigning the minimal frequency value (collected in the corpus) to a declaration that does not appear in the corpus. The declaration unigram score is equal to the logarithm of declaration probability.

9. Expression Score

We use the (partial) expression score, $\mathsf{score(e)}$, to guide the synthesis algorithm and to rank the final expressions (see Section 6.2). For a given (partial) expression e we calculate $\mathsf{score(e)}$ as

$$\mathsf{score}_{pcfg}(\mathsf{e}) + \mathsf{score}_{decls}(\mathsf{e}) - \mathsf{pen}_{rep}(\mathsf{e}) - \mathsf{pen}_{dis}(\mathsf{e})$$

The PCFG score, $\mathsf{score}_{pcfg}(\mathsf{e})$, is proportional to product of the probabilities from the production rules used to build e:

$$\mathsf{score}_{pcfg}(\mathsf{e}) = \mathsf{c}_{pcfg} \cdot \log\left(\prod_{\mathsf{r} \in \mathsf{Rules(e)}} \mathsf{prob(r)}\right)$$

The score $\mathsf{score}_{decls}(\mathsf{e})$ is equal to the sum of the scores of all declarations used to build e:

$$\mathsf{score}_{decls}(\mathsf{e}) = \sum_{\mathsf{d} \in \mathsf{Decls(e)}} \mathsf{score}_d(\mathsf{d})$$

The repetition penalty, $\mathsf{pen}_{rep}(\mathsf{e})$, penalizes all repeated and unnecessary declarations in e:

$$\mathsf{pen}_{rep}(\mathsf{e}) = \mathsf{c}_{rep} \cdot \mathsf{numOfReps(e)}$$

The disarrangement penalty, $\mathsf{pen}_{dis}(\mathsf{e})$, penalizes the disarrangement among the local variables that do not appear in the order given in the input text:

$$\mathsf{pen}_{dis}(\mathsf{e}) = \mathsf{c}_{dis} \cdot \mathsf{numOfSwaps(e)}$$

First, the functions Rules and Decls return the production rules and declarations used in e, respectively. Second, the overloaded function prob returns the probability of the rule r or the declaration d. Third, the function $\mathsf{numOfReps}$ returns the number of the repeated declarations in e. Finally, the function $\mathsf{numOfSwaps}$ calculates the number of the swaps we need to perform between the local variables in e to arrange them as in the input text. We learn the values of the coefficients c_{pcfg} and c_{rep}, shown in Figure 12, using the method in Section 11.1. Note that we manually set the coefficient c_{dis} to a small real value.

10. Building PCFG and Unigram Models

We build both unigram and PCFG models by analyzing and extracting data from the GitHub Java corpus [1] that contains

	Input	Output	Rank			Time
			NoPU	NoP	All	[ms]
1	copy file fname to destination	FileUtils.copyFile(new File(fname), new File(destination))	>10	>10	1	62
2	does x begin with y	x.startsWith(y)	>10	>10	1	63
3	load class ”MyClass.class”	Thread.currentThread().getContextClassLoader() .loadClass(”MyClass.class”)	>10	>10	2	46
4	make file	new File(<arg>).createNewFile()	>10	>10	1	63
5	write ”hello” to file ”text.txt”	FileUtils.writeStringToFile(new File(”text.txt”), ”hello”)	>10	>10	1	62
6	readFile(”text.txt”,”UTF-8”)	FileUtils.readFileToString(new File(”text.txt”), ”UTF-8”)	>10	>10	5	47
7	parse ”2015”	Integer.parseInt(”2015”)	>10	>10	1	16
8	substring ”OOPSLA2015” 6	”OOPSLA2015”.substring(6)	>10	>10	1	46
9	new buffered stream ”text.txt”	new BufferedReader(new InputStreamReader(new BufferedInputStream(new FileInputStream(”text.txt”))))	>10	1	1	63
10	get the current year	new Date().getYear()	>10	>10	6	125
11	current time	System.currentTimeMillis()	1	1	1	31
12	open connection ”http://www.oracle.com/”	new URL(”http://www.oracle.com/”).openConnection()	>10	>10	1	31
13	create socket ”http://www.oracle.com/” 80	new Socket(”http://www.oracle.com/”, 80)	>10	>10	5	47
14	put a pair (”Mike”,”+41-345-89-23”) into a map	new HashMap().put(”Mike”, ”+41-345-89-23”)	>10	9	1	125
15	set thread max priority	Thread.currentThread().setPriority(Thread.MAX_PRIORITY)	1	>10	1	93
16	set property ”gate.home” to value ”http://gate.ac.uk/”	new Properties().setProperty(”gate.home”, ”http://gate.ac.uk/”)	>10	>10	2	94
17	does the file ”text.txt” exist	new File(”text.txt”).exists()	>10	4	1	62
18	min 1 3	Math.min(1, 3)	>10	9	1	31
19	get thread id	Thread.currentThread().getId()	1	1	1	47
20	join threads	Thread.currentThread().join()	>10	1	2	16
21	delete file ”text.txt”	new File(”text.txt”).delete()	>10	1	1	62
22	print exception ex stack trace	ex.printStackTrace()	>10	>10	7	47
23	is ”text.txt” directory	new File(”text.txt”).isDirectory()	>10	>10	1	47
24	get thread stack trace	Thread.currentThread().getStackTrace()	1	1	1	47
25	read line by line file ”text.txt”	FileUtils.readLines(new File(”text.txt”))	>10	8	2	78
26	set time zone to ”GMT”	Calendar.getInstance().setTimeZone(TimeZone.getTimeZone(”GMT”))	>10	>10	1	62
27	pi	Math.PI	2	1	1	0
28	split ”OOPSLA-2015” with ”-”	”OOPSLA-2015”.split(”-”)	>10	>10	1	31
29	memory	Runtime.getRuntime().freeMemory()	3	2	1	16
30	free memory	Runtime.getRuntime().freeMemory()	2	4	1	15
31	total memory	Runtime.getRuntime().totalMemory()	2	2	1	32
32	exec ”javac.exe MyClass.java”	Runtime.getRuntime().exec(”javac.exe MyClass.java”)	>10	1	1	15
33	new data stream ”text.txt”	new DataInputStream(new FileInputStream(”text.txt”))	>10	>10	4	47
34	rename file ”text1.txt” to ”text2.txt”	new File(”text1.txt”).renameTo(new File(”text2.txt”))	>10	>10	1	47
35	move file ”text1.txt” to ”text2.txt”	FileUtils.moveFile(new File(”text1.txt”), new File(”text2.txt”))	>10	>10	1	62
36	concat ”OOPSLA” ”2015”	”OOPSLA”.concat(”2015”)	>10	9	1	63
37	read utf from the file ”text.txt”	new DataInputStream(new FileInputStream(”text.txt”)).readUTF()	>10	>10	7	46
38	java home	SystemUtils.getJavaHome()	2	1	1	32
39	upper(text)	text.toUpperCase()	>10	>10	1	31
40	compare x y	x.compareTo(y)	>10	>10	1	15
41	BufferedInput ”text.txt”	new BufferedInputStream(new FileInputStream(”text.txt”))	>10	>10	1	32
42	set thread min priority	Thread.currentThread().setPriority(Thread.MIN_PRIORITY)	1	1	1	93
43	create panel and set layout to border	new Panel().setLayout(new BorderLayout())	>10	1	1	156
44	sort array	Arrays.sort(array)	>10	>10	1	31
45	add label ”Names:” to panel	new Panel().add(new Label(”Names:”))	>10	>10	1	78

Figure 8. The table that shows the examples we use to tune *anyCode*.

over 14,500 Java projects containing over 1.8 million files. The corpus includes the source files from the projects forked at least once. Although GitHub contains over 600,000 Java repositories, this criteria provides a quality corpus, filtering out less popular repositories.

We decide to analyze each Java source file individually to reduce analysis time. We are aware that building an entire project using standard Java compiler results in a more accurate model. However, this brings two problems: 1) long analysis time, and 2) non-standardized building, using various scripts and resources if they are uploaded. The latter usually includes manual intervention which becomes completely infeasible when building over 14,500 Java projects. Although this reduces the quality of the model, it is compensated by the fact that we can analyze many more projects.

We use Eclipse JDT parser [10] to parse each file. To improve the model we build our own symbol table and type-checker. The symbol table identifies API declarations in an expression and the type-checker checks if the expression type-checks against them. For each API declaration that we find in corpus, we extract both the occurrence frequency (to build unigram model) and a multi-composition and its frequency (to build PCFG model). For each local variable and literal used in the corpus, we abstract away names and keep their types to improve PCFG model.

11. Evaluation

We first present a set of examples we use to tune *anyCode* and explain our tuning technique. Next, we present a set of examples we use to evaluate *anyCode* together with the experimental results.

We ran all experiments on a machine with a quad-core processor with 2.7Ghz clock speed and 16MB of cache. We imposed a 8GB limit for allowed memory usage. Software configuration consisted of Windows 7 (64-bit) and Java(TM) Virtual Machine 1.7.0.55. We also set *anyCode* parameters N and M to 10 and S to 5. By setting the small number of

	Input	Output	Rank			Time
			NoPU	NoP	All	[ms]
1	write 2015 to data ouput stream ”text.txt”	new DataOutputStream(new FileOutputStream(”text.txt”)).write(2015)	>10	>10	>10	62
2	get date when file ”text.txt” was last time modified	new Date(new File(”text.txt”).lastModified()).getTime()	>10	>10	2	219
3	check file ”text.txt” ”read” permission	AccessController.checkPermission(new FilePermission(”text.txt”, ”read”))	>10	>10	1	62
4	read lines with numbers from file ”text.txt”	new LineNumberReader(new InputStreamReader(new FileInputStream(”text.txt”))).readLine()	>10	>10	5	78
5	StreamTokenizer(”text.txt”)	new StreamTokenizer(new BufferedReader(new FileReader(”text.txt”)))	>10	>10	6	16
6	read from console	new BufferedReader(new InputStreamReader(System.in)).readLine()	>10	>10	7	31
7	is file ”text.txt” data available	new DataInputStream(new FileInputStream(”text.txt”)).available()	>10	>10	1	78
8	SequenceInputStream(”text1.txt”, ”text2.txt”)	new SequenceInputStream(new FileInputStream(”text1.txt”), new FileInputStream(”text2.txt”))	>10	>10	>10	31
9	get double value x	Double.valueOf(x).doubleValue()	>10	>10	>10	47
10	write object o to file output stream ”data.obj”	new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(”data.obj”))).writeObject(o)	>10	>10	1	78
11	1 xor 5	new BitSet(1).xor(new BitSet(5))	>10	>10	>10	16
12	create bit set and set its 5th element to true	new BitSet(5)	>10	5	1	156
13	accept request on port 80	new ServerSocket(80).accept()	>10	>10	5	46
14	ResourceStream(”text.txt”)	ClassLoader.getSystemResourceAsStream(”text.txt”)	>10	>10	1	32
15	gaussian	new Random(System.currentTimeMillis()).nextGaussian()	4	4	3	15
16	thread group	Thread.currentThread().getThreadGroup()	1	1	1	<1
17	create panel and set layout to grid	new Panel().setLayout(new GridBagLayout())	>10	1	1	125
18	get screen size	Toolkit.getDefaultToolkit().getScreenSize()	2	1	1	62
19	get splash screen graphics	SplashScreen.getSplashScreen().createGraphics()	>10	3	3	47
20	get v’s 10th element	v.elementAt(10)	>10	>10	1	63
21	polygon x y	new Polygon(x, y, x.length)	>10	>10	5	15
22	dialog ”Welcome!”	JOptionPane.showMessageDialog(null, ”Welcome!”)	>10	>10	1	31
23	get display refresh rate	GraphicsEnvironment.getLocalGraphicsEnvironment() .getDefaultScreenDevice().getDisplayMode().getRefreshRate()	>10	>10	1	63
24	make obj’s string using reflection	ToStringBuilder.reflectionToString(obj)	1	5	1	125
25	get obj’s hash code using reflection	HashCodeBuilder.reflectionHashCode(obj)	3	4	1	109
26	are x and y equal with respect to reflection	EqualsBuilder.reflectionEquals(x, y)	>10	>10	7	109
27	get keystroke modifiers	KeyEvent.getKeyModifiersText(keystroke.getModifiers())	>10	>10	3	47
28	generate ”RSA” private key	KeyPairGenerator.getInstance(”RSA”).generateKeyPair().getPrivate()	4	8	1	31
29	get ”MyClass.class” source code	Class.forName(”MyClass.class”).getProtectionDomain().getCodeSource()	>10	>10	5	62
30	new x instance	Class.forName(x).newInstance()	>10	>10	>10	32
31	add mouse press to robot	robot.mousePress(InputEvent.BUTTON1_MASK)	>10	>10	6	46
32	reverse list	Collections.reverse(list)	>10	>10	>10	32
33	convert prop	ExtendedProperties.convertProperties(prop)	1	1	1	15
34	intersection of rectangle 4 5 with rectangle 3 2	new Rectangle(5, 4).intersection(new Rectangle(3, 2))	>10	>10	2	78
35	set cursor over label to hand	label.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR))	>10	>10	1	78
36	read big integer from console	new Scanner(System.in).nextBigInteger()	>10	>10	6	47
37	delete file ”text.txt” when JVM terminates	new File(”text.txt”).deleteOnExit()	>10	4	2	47
38	if blank(x) ”2015” else x	StringUtils.defaultIfBlank(x, ”2015”)	>10	>10	5	47
39	get date instance for Germany	DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM, Locale.GERMANY)	>10	>10	1	46
40	processors	Runtime.getRuntime().availableProcessors()	3	2	1	<1
41	set command ”enable” to formatted text field x	new JFormattedTextField(x).setActionCommand(”enable”)	>10	>10	1	156
42	does map include value 1	map.containsValue(1)	>10	>10	>10	63
43	move x to y	FileUtils.moveFile(x, y)	>10	>10	1	47
44	writeBytes(bytes, fname)	FileUtils.writeByteArrayToFile(new File(fname), bytes)	>10	>10	2	31
45	new horizontal slider 0 50 25	new JSlider(JSlider.HORIZONTAL, 0, 50, 25)	>10	>10	>10	78

Figure 9. The table that shows the results of the comparison of the different *anyCode* configurations.

synthesis steps we intend to limit *anyCode*’s execution time and make it responsive.

11.1 Tuning Technique

The goal of the tuning is to learn the coefficients c_x for the declaration and the expression scores. For this purpose, we wrote 45 examples, shown in Figure 8. Each example consists of a textual description and local variables as input (column Input) and an expected expression as output (column Output). We say that *anyCode* successfully synthesizes the expected expression exp if, for a given textual description, *anyCode* lists exp among the top N synthesized expressions.

Let us formally define what our tuning objectives are. For a given example e, let Rank be the function that returns the rank of the expected expression. If *anyCode* cannot synthesize the expression Rank returns $+\infty$. Now, let us define the indicator function IN and the function FN as follows:

$$\mathsf{IN}(\mathsf{e}) = \begin{cases} 1 & \mathsf{Rank}(\mathsf{e}) \leq \mathsf{N} \\ 0 & \mathsf{Rank}(\mathsf{e}) > \mathsf{N} \end{cases}$$

$$\mathsf{FN}(\mathsf{e}) = \begin{cases} \mathsf{Rank}(\mathsf{e}) & \mathsf{Rank}(\mathsf{e}) \leq \mathsf{N} \\ 0 & \mathsf{Rank}(\mathsf{e}) > \mathsf{N} \end{cases}$$

Then, our objective functions are as follows:

$$\mathsf{S}(\mathsf{E}) = \textstyle\sum_{\mathsf{e}_i \in \mathsf{E}} \mathsf{IN}(\mathsf{e}_i) \qquad \mathsf{R}(\mathsf{E}) = \textstyle\sum_{\mathsf{e}_i \in \mathsf{E}} \mathsf{FN}(\mathsf{e}_i)$$

The function S(E) returns the number of successfully synthesized expected expressions for some set of examples E. R(E) returns the sum of ranks, but only ranks that are smaller than N. Given those functions and the set of tuning examples E, our goal is to maximize S(E) and minimize R(E).

To efficiently tune the system, we use the bounded approach, where for each coefficient we define the set of possible values (typically ten equidistant values between zero and one). Then, for some limited number of steps, the tuning algorithm 1) randomly chooses a coefficient, 2) iterates through all values of the value set for the coefficient, 3) runs for each value *anyCode* on the entire set of training examples, and 4) keeps the value that satisfies objectives the best. While iterating through the values for the chosen coefficient, the other coefficients are set to their best values. To avoid local maximums and minimums we rerun the entire algorithm several times. Each time, we set initial coefficients' values randomly, using values from the value sets. Finally, we compare the results of each run and keep the configuration that maximizes S(E) the most. If there are many such configurations, we choose the one with the smallest R(E) result. Using this method, we manage to learn values for the coefficients as shown in Figure 12.

11.2 Measuring Efficiency and Effectiveness

To measure efficiency and effectiveness of *anyCode* we wrote another set of 45 examples as shown in Figure 9. The examples are given in the same form as the tuning examples. Again, we measure the ability of *anyCode* to successfully synthesize expected expressions and to list them among the top 10 solutions.

The results are shown in Figure 9. The Input column represents the textual descriptions, and the Output column represents the expected expressions. The column Rank represent the ranks of the expected expressions after we run *anyCode*. With >10 we mark the case where the expected expression is not among the top ten synthesized expressions. The Rank column is split in three sub-columns. Each column relates to a different *anyCode* configuration. The first, NoPU, denotes *anyCode* that does not use unigram and PCFG models. In this setting all declarations have the same unigram score and all PCFG productions have the same probability. The second, NoP, denotes *anyCode* that uses unigram but not PCFG model. In this setting, the tool uses unigram model to select declarations, but all PCFG productions still have the same probability. The last, All, denotes *anyCode* that uses both unigram and PCFG to guide the synthesis algorithm and to rank expressions. The results show that the system without both models generates only 8 (18%) expected expressions among the top ten solutions and the system with unigram model generates 12 (27%). The system with both models recovers 37 (82%) expected expressions and in 20 (44%) examples the expected expression appears as the top solution. Finally, the column Time shows the times needed to synthesize the top ten expressions for the *anyCode* with both models turned on. All times are between 1 and 219 milliseconds, with an average of 60 milliseconds.

In summary, the results show that *anyCode* can efficiently synthesize the expressions in a small period of time (in less that 220 milliseconds).

12. Limitations

The limitations of our tool are partly a reflection of the vast gap between free-form text including English phrases and Java code that needs to compile correctly.

The first limitation is related to our set of examples. While fairly large by the standards of previous literature, it may not be representative of general results. This limitation comes from the fact that there is no standardized set of benchmarks for the problem that we examine. A parallel corpus with free-form queries as input and desired declarations and expressions as output would be ideal for configuring the parameters and performing the evaluation, yet no such corpus exists.

The second limitation is related to the complexity of the code snippets we synthesize. It comes from the fact that we synthesize only expressions, excluding local variable declarations and other statements. This means that we do not generate control flow constructs like loops and conditional statements (that said, *anyCode* performs copy propagation when applicable to better interpret the corpus when constructing PCFG, which captures data-flow). To support the synthesis of control-flow constructs we might need to use a similar approach to Macho [7], which combines natural language input with input-output examples, or to combine our work with approaches such as Prime [25], which suggest temporal order among declarations and naturally fits in the context of IDE-based code synthesis. We believe that such approaches may lead to efficient synthesis of code fragments whose complexity goes up to the complexity of method bodies.

13. Related Work

We start by discussing related work that combines NLP and program synthesis techniques, then present program synthesis tools with similar goals as *anyCode*, and, finally, compare *anyCode* with those approaches and tools.

SmartSynth [21] generates smartphone automation scripts from natural language descriptions. It uses NLP techniques to infer components and their partial dataflow from NL description. Next, it uses type based synthesis to construct the scripts. Macho [7] transforms a natural language description into a simple program using a natural language parser, corpus, and input-output examples. It maps English into database queries, then selects the candidate solutions, combines them, and tests them using input-output examples. Little and Miller [22] built a system that translates a small number of keywords, provided by the user, into a valid expression. It selects declarations using the keywords. Next, it combines them trying to increase the number of the keywords that appear in declarations. SNIFF [6] uses natural language to search for code examples. It collects a corpus, code examples, and uses API documentation to annotate the examples and method calls with keywords. NaturalJava [26] allows a user to create and manipulate Java programs using natural language input. It requires the user to think and explicitly describe commands at the syntactical level of Java.

Metafor [23] transforms a story (in natural language) into a program template. It tries to obtain program structure by interpreting nouns as program objects, verbs as functions and adjectives as properties. Several tools [28, 31] search and synthesize code fragments using input-output examples. Another tool [17] uses genetic programming approach with different degrees of human guidance which includes names of library functions and test cases. The tool grows a new functionality using user suggestions and grafts it into the existing code.

InSynth [16] asks a user to specify the desired type and produces a set of ranked expressions, instances of the desired type. InSynth ranks the solutions based on the declaration unigram model. SLANG [27] takes a program with holes and produces the most likely completions, sequences of method calls. It uses an N-gram language model to predict and synthesize a likely method invocation sequence as well as method arguments. CodeHint [12] is a dynamic synthesis tool that uses a runtime information and unigram model to generate and filter candidate expressions. A user provides tests and specification and the tool generates candidates and checks them against the tests and specification. XSnippet [29] takes a user query to extract Java code from the sample repository. XSnippet ranks solutions based on their length, frequency, as well as context sensitive and independent heuristics. The user needs to initiate additional queries to fill in the method arguments. Strathcona [18] automatically extracts a query based on the structure of the developed code. It does not allow a user to explicitly describe their needs. PARSEWeb [32] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. Several code completions tools [4, 9] propose declarations and code templates. Both systems use API declaration call statistics from the existing code examples to present solutions with appropriate statistical confidence value. Unlike the tools mentioned in this paragraph, *anyCode* uses textual description, PCFG and unigram models to synthesize meaningful expressions.

anyCode has a number of differences with the previous tools mentioned above. First, the mentioned tools that use textual input apply a simple mapping model which maps verbs to methods (actions) and nouns to arguments (objects). We observe that such a mapping model can be further refined, for example, to support the observation that both verbs and nouns appear in method names and can be used to select methods. For this reason, we build the model described in Sections 4, 5 and 8 that 1) uses NLP tools to slice the input into word groups and extract words from declarations; 2) takes into account the type and position of words, in input and in declarations to calculate the matching score; 3) uses unigram model to select the most popular declarations. Second, to our knowledge we are the first to build the related world map that maps English words to the related API words and use it successfully. The purpose of this map is to make the input more flexible. To build the map we use the novel technique, described in Section 7, which discovers the most relevant word relations in WordNet in the "programming/API" domain. Third, we use the PCFG model, extracted from the large code corpus, to guide the synthesis algorithm and rank solutions, as described in Section 6. To the best of our knowledge, we are the first to use PCFG model in the code synthesis context.

14. Conclusions

We presented *anyCode*, a tool for code synthesis that combines unique flexibility in both its input and output. On one hand, *anyCode* performs parsing of the free-form text input that may contain a mixture of English and code fragments. On the other hand, *anyCode* automatically constructs valid Java expressions for a given program point and is able to generate combinations of methods not encountered previously in the corpus. Ensuring this flexibility required a new combination of natural language processing, mapping text to code elements, and code synthesis based on probabilistic grammar. Our experience with evaluating tool on 45 diverse examples suggests that there are a number of scenarios in which such functionality can be useful for developers.

Acknowledgments

We thank Rastislav Bodik, Armando Solar-Lezama, Darko Marinov and Ravichandhran Kandhadai Madhavan for valuable discussions and comments.

This work is supported in part by the European Research Council (ERC) project "Implicit Programming".

References

[1] M. Allamanis and S. Charles. Mining Source Code Repositories at Massive Scale using Language Modeling. In *Conf. Mining Software Repositories*. IEEE, 2013.

[2] AnyCode Source Code. `https://github.com/tihomirg/nlpcoder/tree/noola`, 2015.

[3] BitBucket. BitBucket repository hosting service, `https://bitbucket.org/`, 2015.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222, 2009.

[5] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Math., 2009.

[6] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. In *FASE*, pages 385–400, 2009.

[7] A. Cozzie and S. T. King. Macho: Writing programs with natural language and examples. Technical report, University of Illinois at Urbana-Champaign, 2012.

[8] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.

[9] Eclipse Code Recommenders. http://www.eclipse.org/recommenders/, 2015.

[10] EclipseJDT. EclipseJDT, http://www.eclipse.org/jdt/, 2015.

[11] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[12] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663, 2014.

[13] GitHub. GitHub repository hosting service, https://github.com/, 2015.

[14] T. Gvero and V. Kuncak. Interactive synthesis using free-form queries. In *International Conference on Software Engineering, Demo Papers (ICSE Demo)*, 2015.

[15] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *Computer Aided Verification (CAV) Tool Demo*, 2011.

[16] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[17] M. Harman, Y. Jia, and W. B. Langdon. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *SSBSE Challenge Track*, 2014.

[18] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, 2005.

[19] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, 2 edition, 2008.

[20] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quart.*, 2:83–97, 1955.

[21] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[22] G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, pages 84–93, 2007.

[23] H. Liu and H. Lieberman. Metafor: Visualizing stories as code. In *IUI*, pages 305–307, 2005.

[24] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[25] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, pages 997–1016, 2012.

[26] D. Price, E. Riloff, J. L. Zachary, and B. Harvey. NaturalJava: A natural language interface for programming in Java. In *IUI*, pages 207–211, 2000.

[27] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, page 44, 2014.

[28] S. P. Reiss. Semantics-based code search. In *ICSE*, 2009.

[29] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA*, 2006.

[30] SourceForge. SourceForge source code repository, http://sourceforge.net/, 2015.

[31] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM TOSEM*, 23:26:1–26:45, June 2014.

[32] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *ASE*, 2007.

[33] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL*, 2003.

A. Appendix

A.1 Declaration Search Algorithm

The algorithm that searches for the most relevant declarations is shown in Figure 10. From each word group it takes words and selects declarations which contain the words. Then, for every selected declaration it calculates the score. Finally, for every word group it forms declaration groups.

```
fun declSearch(WordGroups, API, Unigram, M)
 size ← WordGroup.length
 DeclarationGroups ← new DeclarationGroup[size]
 foreach(i = 1 to size)
   WordGroup ← WordGroups[i]
   DeclarationGroup ← ∅
   foreach(w ∈ WordGroup)
     DeclarationGroup ← WordDeclarationMap(w)
     foreach(d ∈ DeclarationGroup)
       setScore(dscore(d, wordGroup, Unigram)))
   DeclarationGroups[i] ← keepBest(DeclarationGroup, M)
return DeclarationGroups
```

Figure 10. The method that selects the most likely set of declarations based on input words.

A.2 Expression Synthesis Algorithm

Formally, we define a partial expression pe using the following grammar:

```
pe ::= [pe.] d [(pe, ..., pe)] | literal | local | hole | c(d,PEG)
hole :: = ?Decl(d) | ?Lit(type) | ?Var(type)
type ∈ STypes     literal ∈ Literals   local ∈ Locals
d ∈ Decls    PEG ∈ PEGS
```

First, the partial expression pe might contain a declaration d, which can be preceded by another partial expression (receiver) and can also be followed by a list of partial expressions (arguments). Second, a partial expression can be either a literal (string, boolean or number) or a local variable that a user provides at input. Third, a partial expression can be a hole or a connector.

The algorithm takes an array of declaration groups DGS and PCFG and returns the ranked list of N best expressions, described in synth function, in Figure 11. It is executed in two phases. In the first phase, for each declaration group DG , we take the declaration d ∈ DG and invoke function unfold. The function unfold takes the declaration, PCFG and returns

```
fun synth(DGS, PCFG, S, N)
  //Unfold declaration arguments
  size ← DGS.length;   PEGS ← new PES[size]
  foreach(i = 1 to size)
    DG ← DGS(i)
    foreach(d ∈ DG)
        PEGS[i]← PEGS[i] ∪ unfold(d,i,PCFG,DGS,PEGS,S,N)
    PEGS[i] ← keepBest(N, PEGS[i])
    //Merge partial expressions
  foreach(PEG ∈ PEGS)
    foreach(pe ∈ PEG) Snippets ← merge(pe, S, N)
  return keepBest(N, Snippets)
```

Figure 11. The synthesis algorithm.

N partial expressions that have d as the topmost declaration. To limit the execution time we introduce a bound S on the maximal number of steps in unfold. We collect all partial expressions generated from one declaration group DGS[i] and put them in a partial expression group PEGS[i]. In the end we pass all partial expression groups to the second phase.

In the second phase, for each partial expression pe, from the partial expression group PEG, we call merge method. It uses connectors to connect pe with partial expressions in other groups. We collect all expressions that merge returns and present only the top N to the user. To find the top expressions we use score(pe) introduced in Section 9.

```
for unfold(d, i, PCFG, DGS, PEGS, S, N)
  PESS ← new PES[S+1]
  PESS[1] ← {?Decl(d)}
  for(j = 1 to S)
    Step ← ∅
    foreach(pe ∈ PESS[j])
      hole ← findFirstHole(pe)
      if (comp(hole, i, DGS, PEGS) ∧ j>1)
        //postpone unfolding till merge phase
        foreach(PEG ← findAllComp(hole, i, DGS, PEGS))
          Step←Step ∪ applySub(pe,hole→c(decl(hole),PEG))
      else
      if (∃ prod ∈ PCFG s.t. hole = leftSide(production))
        //unfold using PCFG
        foreach(prod ∈ PCFG(hole))
          Step ← Step ∪ applySub(pe, hole → righSide(prod))
    PESS[j+1] ← keepBest(Step, N)
  return keepWithoutHoles(PESS)
```

We explain in more depth unfold and merge methods. First, the method unfold wraps the declaration d into the hole ?Decl(d). We use PESS as a working list to store the results. It is an array of partial expression sets with length S+1 which denote the number of the steps we will perform in unfold. In each step, the method takes the partial expression pe from PESS[j], calculates new partial expressions by unfolding pe hole, and stores them into PESS[j+1]. hole either becomes a connector or is substituted by the right-hand side of a production prod ∈ PCFG(hole).

We substitute hole with the connector if hole = ?Decl(d) and d is in one of DGS[k], where $k \neq i$. This is checked by the method comp. If the condition is true, we substitute the hole with a connector c(d, PEG) (decl(hole) returns a declaration of a declaration hole). If the condition is not satisfied, we check if there are appropriate productions in PCFG for the hole. PCFG contains the production for a given hole if the production's left-hand side matches with the hole. For each such a production we use its right-hand side as a partial expression that substitutes the hole. Using this method, we create new partial expressions and leave only the top N with the highest score. Also, note that we do not substitute a hole with a connector in the first iteration (condition j>1). This condition allows us to unfold each initial hole at least one in the first phase of synth. In the second phase, this allows us to replace each connector with a partial expression.

```
for merge(pe, NeighborsMap, S, N)
  PESS ← new PES[S+1]
  PESS[1] ← {pe}
  for(i = 1 to S){
    Step ← ∅
    for(pe ∈ PES[i])
      c(d, PEG) ← findFirstConnector(pe)
      for(cpe ∈ findAll(PEG, d))
        Step ← Step ∪ applySub(pe, c(d, PEG) → cpe)
    PESS[i+1] ← keepBest(Step, N)
  return keepBestWithoutConnectors(PESS, N)
```

The method merge is similar to unfold. The main difference is that we substitute the connectors with partial expressions instead of unfolding holes using PCFG. Thus, in each step, we find a connector in a partial expression, then substitute it with the corresponding partial expression from PEG that starts with d. Finally, we return the first N expressions without connectors.

Coefficient	Value	Coefficient	Value
c_m	0.7	c_u	0.3
c_p	0.7	c_s	0.3
c_r	0.9	c_{dis}	10^{-6}
c_{pp}	0.7	c_{ss}	0.5
c_{sp}	0.3	c_{ps}	0.4
c_{pcfg}	0.6	c_{rep}	0.9

Figure 12. The learned values of *anyCode*'s coefficients and the value of c_{dis} that we set manually.