Improving Keyword Programming by Using Neural Text Generation

Author Name: Shu Aochi

Student Number: 18M30410

Graduation Year: 2020

Supervisor: Masuhara Hidehiko

A thesis presented for the degree of
Master of Science

Department Name: School of Computing
University Name: Tokyo Institute of Technology

Date: February 20th, 2020

Acknowledgements

Firstly, I want to thank my parents for supporting me in the last three years. I would also like to thank my supervisor, Professor Masuhara, for advising me patiently. Finally, I would like to thank the members of my lab and other friends in Japan, who makes me remember the memorable experience in Tokyo.

Abstract

Keyword programming is a feature of programming editors that recommends code fragments suitable to the keywords given by the programmer and the surrounding program text around the cursor position. It can be considered as advanced code completion by taking the programming intention through keywords. However, the existing keyword programming algorithm does not consider the meaning of the context, which should be useful in selecting recommendations. In this study, we improve the ranking algorithm by incorporating the likeliness factor of the code fragment concerning the context For estimating the likeliness, we use a neural network-based sentence generator. We implement it as a form of a plug-in on Eclipse and evaluate it by recommending expressions that is hard to obtain from the existing keyword programming.

Contents

1 Introduction **1**
1.1 Programming Environment 1
1.1.1 Efficient Tools of Programming Environment 1
1.2 Code Assist 3
1.2.1 Code Snippets Completion 4
1.2.2 Code Recommendation 5
1.3 Evaluation Criteria 6
1.3.1 Accuracy 6
1.3.2 Precision 8
1.4 Purpose 8
1.5 Overview 9

2 Background **10**
2.1 Keyword Programming 10
2.1.1 Input 10
2.1.2 Extraction 10
2.1.3 Generation 14
2.1.4 Advantages 15
2.2 Neural generation 16
2.2.1 Neural networks 16
2.2.2 RNN: Recurrent Neural Networks 18
2.2.3 LSTM: Long Short Term Memory 20
2.2.4 Neural Text Generation 21

3 Problem **23**
3.1 Difficult to generate a complicated expression 23
3.1.1 Generate duplicated expressions 23
3.1.2 Score function limits long generation 24
3.2 Recommendations are not likely to be used 24

4 Proposal **26**
4.1 Limit the amount of generation 26
4.2 Modify the ranking algorithm 27

5 Implementation **29**
5.1 Input 30
5.2 Extraction 30
5.2.1 ASTParser 30
5.2.2 Java Model 34
5.3 Generation 37
5.4 Neural Generation 41
5.4.1 LSTM model 41
5.4.2 Connect by Socket 41

6 Evaluation **43**
6.1 Tasks 43
6.2 Result 43
6.3 Discussion 44
6.3.1 Method v.s. Variable 44
6.3.2 Speed 44
6.3.3 Accuracy&Precision 45
6.3.4 Input priority 45

7 Related Work **47**

8 Conclusion **48**

Chapter 1
Introduction

1.1 Programming Environment

For a beginner who wants to learn a new programming language, the first thing that they would do is to choose a programming environment, after they decide which programming language to use.

For each programming language, there are always plenty of programming environments that supports a programmer with different features. For example, a Java programmer is likely to choose Eclipse or IntelliJ, while a C programmer prefers Visual Studio. And, both of them may use Vim and customize it with different useful plugins such as TabNine.

Selecting a proper programming environment is essential for a programmer when they start to build a project. However, how should a programmer decide which one is suitable to them? A principle to judge which environment is appropriate for a programmer is the efficiency in developing code.

There are a lot of techniques that could help a user to shorten the time costs. Here are some examples in section 1.1.1.

1.1.1 Efficient Tools of Programming Environment

Visualize a program

Some features of a programming environment visualize the program, which aids a programmer to reduce the time to organize their thoughts.

For example, Kanon [Oka et al., 2017] provides a live programming feature that visualizes the layout of a data structure while a programmer is implementing it. It

helps the user to comprehend a complicated data structure while programming.

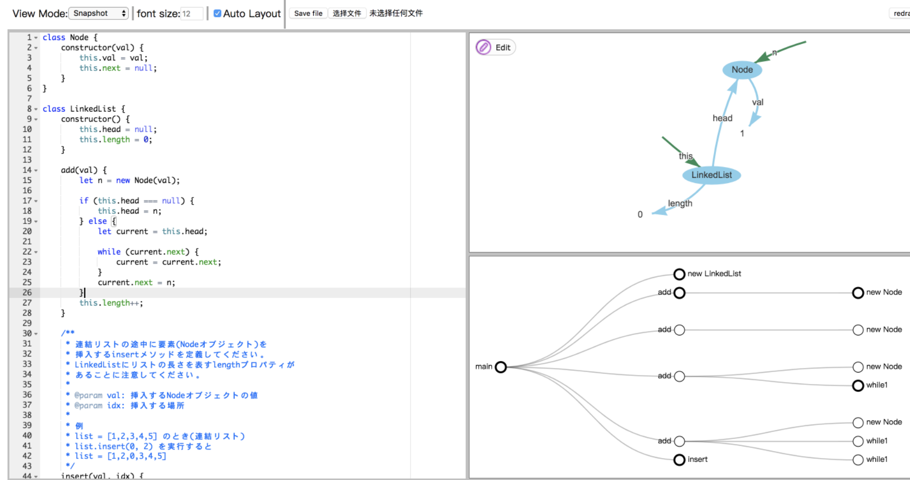


Figure 1.1: A live programming environment: Kanon
(Image from *https://prg-titech.github.io/Kanon/*)

Quick Fix

Moreover, several programming environments advise the user to modify the program when they get stuck. As an illustration, the quick fix feature on Eclipse shown as below offers some options to solve a problem immediately, such as adding a missing package declaration.

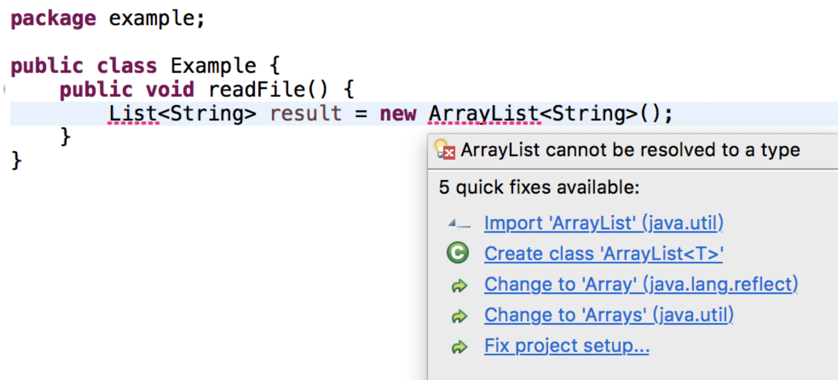


Figure 1.2: Quick Fix of Eclipse provides advice

Template creation

A few programming environments enable a user to avoid writing some code if it is a pattern. For instance, CodeSandbox provides a template for some regular tasks like creating an application. A user does not need to program starting from zero, and it is also pretty comprehensible for a beginner to understand the effect without understanding the whole project. Besides, IntelliJ and Eclipse offers a convenient method to generate a constructor for an object class using its fields.

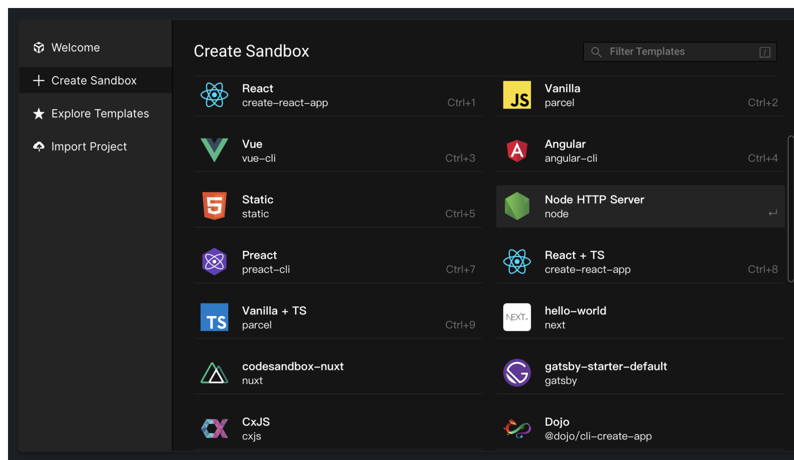


Figure 1.3: Create a template to start to build an application on CodeSandbox (Image from *https://codesandbox.io/*)

In addition, a lot of popular programming environments have a code assist feature that saves the programmers from unnecessary typos. We will discuss more details in the next section.

1.2 Code Assist

Code assist completes the program, given a sequence of keystrokes. This feature is especially vital in saving the user's time when they start to build a complex project. Code assist consists of two primary techniques: code snippets completion and code recommendation.

1.2.1 Code Snippets Completion

Code snippets completion is similar to template creation, which simplifies entering some repeating code patterns. The difference between code snippet completion and template creation is that code snippets need a standard input, and the generated code relates to the input, including an abbreviation, a snippet prefix, and keywords.

Some programming environments such as Eclipse and IntelliJ take an abbreviation or a snippet prefix as input and translate it into a template of code.

For example, when a Java programmer creates a new class and wants to declare the static main method. Instead of typing the whole method, the user only needs to type main and press `Ctrl-Space` on Eclipse or type **psvm** and then press `Tab` on IntelliJ. This technique also helps a novice programmer to learn how to use the programming language without comprehending the whole syntax.

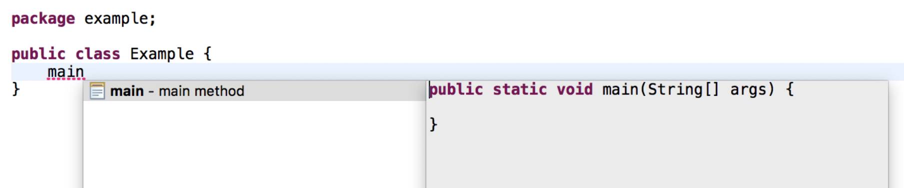


Figure 1.4: `Main` method completion on Eclipse

Unlike common code snippets completion systems, the inputs of NLP2Code [Campbell and Treude, 2017] are a bunch of keywords written in natural language. NLP2Code first provides some choice to completes the keyword query and then insert a code snippet from Stack Overflow in terms of those keywords without leaving the current editor.

Figure 1.5: The flow of NLP2Code completing code
(Image from: Campbell and Treude, 2017)

1.2.2 Code Recommendation

Code recommendation is one of the advanced techniques that assists a programmer in saving time by providing a list of possible code fragments displayed on the editing monitor. The recommendations would show on the editing monitor after a user type some inputs. Then, the user can view the list and select the desired code fragment instead of writing down the whole code.

The information that a code recommendation takes into consideration can be divided into two kinds: explicit and implicit.

Explicit information stands for straightforward information that comes from inputs of a code recommendation system. Similar to code snippets completion, the inputs in such kinds of system vary in different implementations, and there are three categories of input: an abbreviation, a partial expression, and a bunch of keywords.

Han, Wallace and Miller, 2009 and Hu et al., 2019 recommend possible code fragments given an abbreviated input. The former uses a Hidden Markov Model to expand the abbreviation to inputs, while the latter uses a Gaussian mixture model.

Traditional code recommendation systems like the default one on Eclipse provide all possible code fragments after the user inputs a few characters. The recommendations are generally in alphabetical order, and the user could browse the recommendation list and select the required code fragment. For example, when a user wants to write a lengthy method name such as `currentTimeMillis`, instead of writing the whole characters, the user only needs to type **cu**, and then the method `currentTimeMillis`

will appear on the recommendation list.

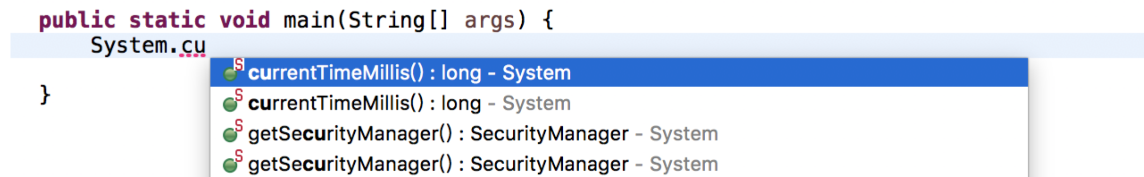


Figure 1.6: Eclipse recommend code based on prefix

Robbes and Lanza, 2008 proposes a code recommendation system that also inputs a few characters, but the recommendations are sorted according to the user's programming history.

In addition, Little and Miller, 2009 provides expressions given some keywords that represent the user's intention. An expression is more likely to show on the list if it contains more words in the keyword query. The programmers, especially who are not familiar with the programming language, do not need to rack their brain to remember the correct method name from a complicated API. The complete procedure of its implementation QUACK is shown in the following figure.

In contrast, implicit information denotes information that can be exploited from the context.

TabNine takes the whole context into consideration, and Bruch, Monperrus and Mezini, 2009 refers to previous method invocations. Both of them arrange the recommendations by their probabilities. The previous system calculates probabilities by a GPT-2 model, and Bruch, Monperrus and Mezini, 2009 utilizes an algorithm named Best Matching Neighbors(BMN) based on the K-Nearest Neighbors algorithm.

1.3 Evaluation Criteria

We define two dimensions to estimate our code completion system and others, which are accuracy and precision.

1.3.1 Accuracy

Accuracy is evaluated by both the occurrence and similarity. Occurrence stands for whether the desired expression is on the recommendation list. And the similarity of two expressions depends on how many tokens are common in both.

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        add line
    }
    return array;
}
```

Ctrl + Space

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        add line
    }
    return array
}
```

```
array.add(src.readLine()) [from Quack]
array.add(System.console().readLine()) [from Quack
new ArrayList<String>().add(src.readLine()) [from Q
Press 'Ctrl+Space' to show Template Proposals
```

Enter

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        array.add(src.readLine());
    }
    return array;
}
```

Figure 1.7: Code recommendation based on keyword programming
(Image from Little and Miller, 2009)

It is apparent that a recommendation is accurate if it appears on the recommendation list. However, why we also define accuracy regarding the similarity? The answer is that an incorrect recommendation sometimes can give the user a hint, which enables them to write the correct code by themselves.

For example, if a user wants to print a result on the monitor in Java, the desired expression should be *System.out.println(result)*. However, for those who are not familiar with this programming language, it is almost impossible for them to write the token **println** without any suggestions. If a similar expression *System.err.println(result)*

```
/**
 * Concatenate two strings
 * @param a the first string to concatenate
 * @param b the second string to concatenate
 * @return the concatenated string
 */
st
static String concat(String a, String b) {    Tab
```

Figure 1.8: TabNine recommends code regarding comments of program

appears on the top of the recommendation list, although it is not a desired one, it can give the user a hint to use the keyword **println**.

Therefore, we estimate the accuracy by both occurrences and similarity.

1.3.2 Precision

Precision is judged by the order where the desired code fragment appears on the list.

It is also essential for a code recommendation system. For example, the code completion system on Eclipse could ensure the desired code fragments shown on the recommendation list, since it generates all possible fragments that are stick to the syntax rule and type rule of the programming language. However, the user has to roll the mouse wheel to find out their desired one when the object class contains hundreds of methods such as `JButton` shown as Figure 1.9 .

1.4 Purpose

In this paper, we aim to build a code recommendation system that:

- Suitable to all programmer from a beginner to an expert.
- Recommend the expression that satisfies the user's purpose.
- Can recommend a complicated expression.
- Be able to give the programmer a hint to write their required expression when it is not on the recommendation list.

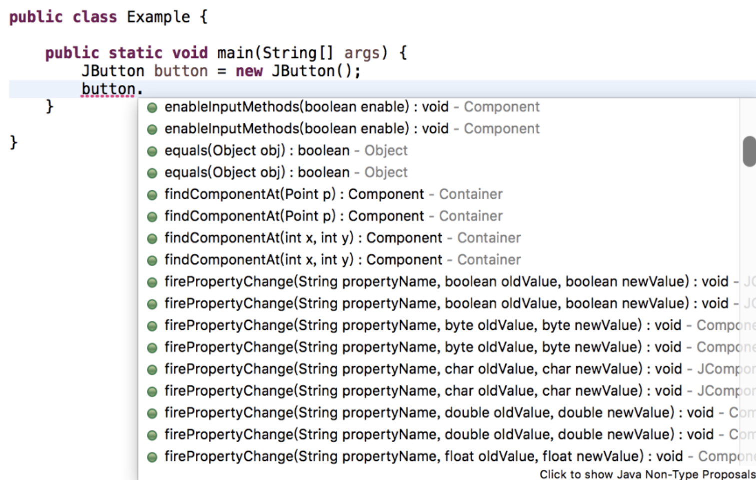


Figure 1.9: Available methods in the class `JButton`

1.5 Overview

This paper is organized as follows.

First, Chapter 1 gives a brief overview of code recommendation and our goal of this paper.

Chapter 2 shows how keyword programming works and why we choose it as our previous work. Then we introduce a neural text generation approach that is able to predict probabilities of the next tokens given the preceding context.

Subsequently, we demonstrate the limitation occurring in previous works in Chapter 3.

In Chapter 4, we put forward our proposal which is to improve keyword programming by exploiting information from context.

Then, we explain the mechanism of our implementation ACKN in Chapter 5. And we evaluate several systems based on the above approach, discuss the results from each experiment, and describe our achievements and limitations in Chapter 6.

We illustrate the benefits and disadvantages of several related research in regards to code snippets completion and code recommendation in Chapter 7.

Our conclusions are drawn in the final chapter.

Chapter 2
Background

2.1 Keyword Programming

Keyword programming is a technique that translates a keyword query to an expression. It is similar to a search engine that gets the desired website by a few keywords. A keyword programming system has three parts: input, extraction, and generation.

2.1.1 Input

Just like its literal meaning, a keyword programming system inputs a bunch of keywords to program.

A keyword query is similar to a piece of pseudo-code or comment that conveys a certain meaning. For example, a keyword query **print f name** stands for an operation to display the name of variable **f** on the monitor.

Moreover, it does not make a difference if the user changes the order of keywords. Which means, a keyword query **print f name** is identical to **print name f** or **f print name**.

Furthermore, for each word in the keyword query, repeated words are only counted once. Considering a keyword query **add number one to number two** as an example. In this query, the word **number** occurs twice. Since a repeated word is only counted once, it is actually the same as **add number one to two**.

2.1.2 Extraction

A keyword programming system needs to model the programming language in order to generate all possible expressions. A model M is defined as a triple (T, L, F),

where T stands for a set of types, L denotes a set of labels and F is short for a set of function.

Type Set: T

The set T consists of all available type's fully qualified name, such as `java.lang.String` or `java.util.List`.

Moreover, *sub*(t) defines the set of sub-types of t. Similar to the type `java.lang.Object` in Java, there is a universal super-type T.

Label Set: L

The name of a variable or a method is represented by a label separated by punctuation or a capital letter. For example, label (instance, of) denotes a method `instanceOf` or $instance_of$.

The reason to apply this is because programmers usually define a variable or a method in a camel case or a snake case.

Function Set: F

A function set includes elements that are modeled from methods, fields, and local variables. And a function is defined as a list of (T, L, T ... T). The initial T stands for the return type, L is the label and the remaining T denotes all the parameter types. For example, a method `String toString(int i)` in Java will be modeled as `(java.lang.String, (to, string), int)`.

In addition, *ret*(f) stands for the return type, *label*(f) represents the label, and *params*(f) is the parameter types of function *f*.

Function Tree

An expression is modeled as a function tree that is made of functions.

Figure 2.1 shows a function tree of the Java expression *System.out.println(result)*. It consists of four nodes. The first one `System` is a class name, then `out` is a field in class *java.lang.System* and returns type *java.io.PrintStream*. The next node `println` is a method that receives type *java.io.PrintStream* and returns type *void*, and the parameter type is *java.lang.String*. The final node is `result`, which returns type *java.lang.String*.

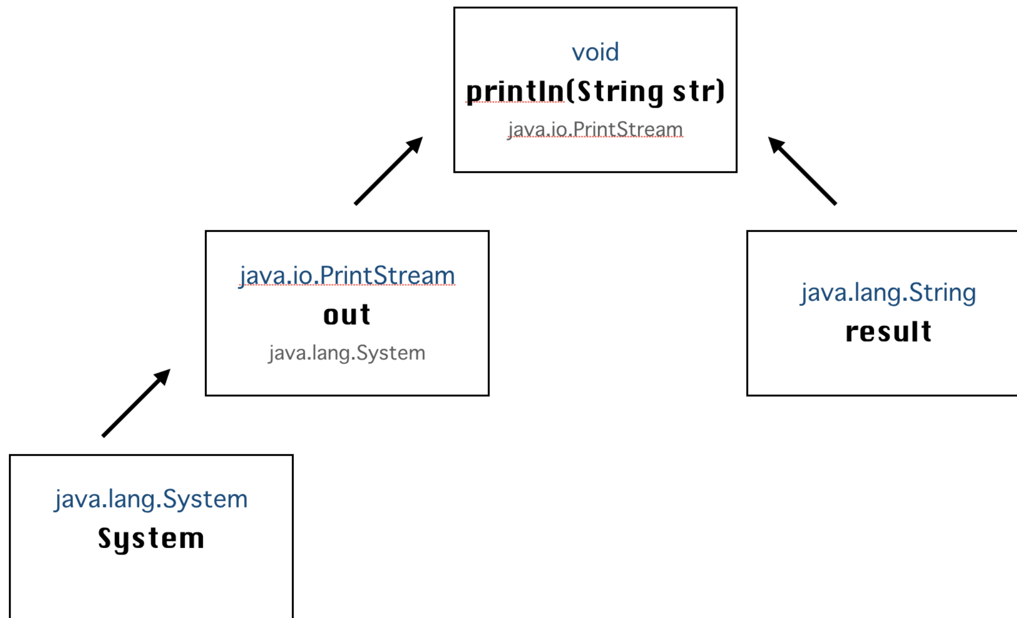


Figure 2.1: Function tree of the expression *System.out.println(result)*

In addition, `depth` stands for the height of a function tree in this paper. For example, a local variable *result* is in the depth of one, and a method invocation expression *list.add()* is in the depth of two.

Except for the methodology above, keyword programming also provides a few ways to model some elements for a specific programming language.

Types

There are three situations to model a type of java: primitive type,generic type and object classes.

In Java, the primitive type has 8 kinds, and each of them is modeled as their corresponding wrapper class. For instance, *int* seems identical to *java.lang.Integer*.

Moreover, the model denotes a generic type literally as it is, which means a type *List<String>* is recognized as a new type. Currently, keyword programming does not completely support a type system.

Finally, an object class or an interface is modeled as its fully qualified name such as *java.io.PrintStream*.

Local Variables

Local variables are represented by the type that its return and the variable name of itself. For example, the local variable *String s* is modeled as `(java.lang.String, (s))`.

Fields

The field is divided into two kinds: static and non-static.

A non-static field is modeled as functions that return the type of the field and take the receive object class as a parameter. For example, a field `String name` of class *Fruit* is modeled as `(String, (name), Fruit)`

By contrast, a static field is usually represented by two functions. One is the same as the non-static field, and another adds the simplified receiver-type name to the label. For instance, the static field `out` in *java.lang.System* could both be modeled as `(java.io.PrintStream, (out), java.lang.System)` and `(java.io.PrintStream, (system, out))`.

Methods

Like a field, methods are also separated by a static one and a non-static one.

A non-static method is modeled in terms of its receiver-type, return-type, parameters' type, and its name. For instance, the method *public void print()* of *java.lang.System* is translated to `(void, (print), java.lang.System)`.

A static method is modeled as two functions. For example, a method *static BigDecimal valueOf(double val)* will become `(java.math.BigDecimal, (value, of), java.math.BigDecimal, double)` or `(java.math.BigDecimal, (big, decimal, value, of), double)`

Instance creation

Instance creations are similar to a method, but only add a keyword **new** to the label set. For instance, a default instance creation for class String could be represented as `(java.lang.String, (new, string), int)`.

2.1.3 Generation

Generation algorithm

The existing keyword programming generates expressions with a recursive algorithm. Figure 2.2 shows how the algorithm generates method invocations that do not have any receivers.

```
1:  procedure GenerateMethodInvocations(t, h)
2:      for 1 ≤ i ≤ h do
3:          for all method where ret(method) ∈ sub(t) do
4:              for all parameter do
5:                  parameterType ← GetParameterType(paramter)
6:                  parameterSet ← GenerateMethodInvocations(t, h − 1)
7:              end for
8:              methodSet ← GenerateMethods(parameterSet)
9:          end for
10:     end for
11:     return methodSet
12: end procedure
```

Figure 2.2: Greedy algorithm for method invocation generation

Score function

Keyword programming arranges generated expressions by a score calculation function. For each expression, there are four rules including:

- For each depth, the score decreases by 0.05
- If the expression contains a keyword that in the keyword query, then the score adds 1.0
- Otherwise, if the word in the expression is not in the keyword query, then the score minuses 0.01.
- If the element is a local variable or a member method, then the score will add 0.001

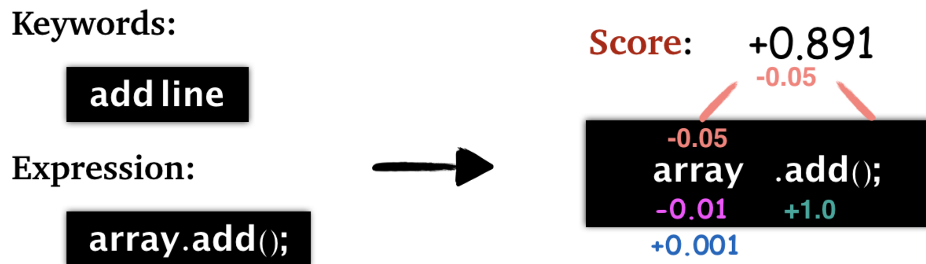


Figure 2.3: Score function calculates *array.add()* given the keyword query **add line**

We will demonstrate the rule by an example of calculating the score for a function array.add() given the keyword **add** and **line**.

First, since the depth of *array.add()* is two, the initial score is **-0.1**.

Then, since the token `array` is not in the keyword query, the score becomes **-0.11**.

Because the token `add` belongs to the keyword query, the score becomes **+0.89**.

Finally, since the token `array` is also a local variable, the final score is **+0.891**.

2.1.4 Advantages

Most of the traditional works are not beginner-friendly. We use Eclipse as an example to illustrate this point. For a programmer who wants to use the code completion system on Eclipse, they need to remember all the spelling correctly of API. Otherwise, they have to search on the internet and seek some examples, which is desperate, especially for a novice programmer.

In contrast, since a keyword programming system takes natural language as inputs, it is easy to use for a programmer no matter whether they are beginners or not.

Keyword programming is generic

Although the existing keyword programming models Java, it does not mean the target programming language should only be Java. As we know, most of the object orient programming languages have similar syntax. As a result, we could also implement the keyword programming on other programming languages such as python.

High accuracy inputting appropriate keywords

In the previous research, the existing keyword programming proves that the accuracy could achieve appropriately 90% if the keyword query is exactly identical to the desired expression. Although the accuracy becomes 50% when the keyword query is provided by a human. We could assume that the keyword query has a better performance when the keyword query contains more words that appear in the desired expression.

Represent programmers' intention

Last but not least, unlike most current code recommendation systems, a keyword programming system is able to represent the programmers' intention precisely. Consequently, the programmer does not need to struggle to remember the method name from API that contains abundant functions.

2.2 Neural generation

Neural text generation is the process of generating a likely token by using neural networks. The generation procedure regards the implicit information from the proceeding sentences exploited from a deep-learning algorithm.

In this chapter, we will introduce several neural networks and the one we chose, which is the long short time memory(LSTM). Then we illustrate how to use such a model to generate a likely token by an example.

2.2.1 Neural networks

It is common to estimate the probability of an event that happened in the future through a statistical model. In the traditional approach, we often use some straightforward concepts from statistics, such as frequency, to forecast what would occur later. However, such kinds of models are inadequate to predict accurately and precisely.

Therefore, a lot of machine-learning algorithm is designed to increase the accuracy and precision by analyzing a massive amount of data.

Deep learning is a subset of machine learning that uses neural networks to simulate how a human being's brain works. A human's brain can translate a piece of music to

a signal and figure out what the music represents by analyzing the signal. A neural network uses different layers to simulate this procedure.

In a neural network, `layer` denotes a collection of neurons that are at a particular depth. A neural network usually has an input layer, an output layer, and several hidden layers. An input layer and an output layer manipulate the input and the result. The hidden layers are the essence of a neural network, which have the capacity to `learn` the data by minimizing an error function.

For example, the Figure 2.4 shows the structure of a neural network. Each circle represents a neuron, and each neuron has an input. The input will pass to another neuron by a particular calculation, and finally deduce to an output.

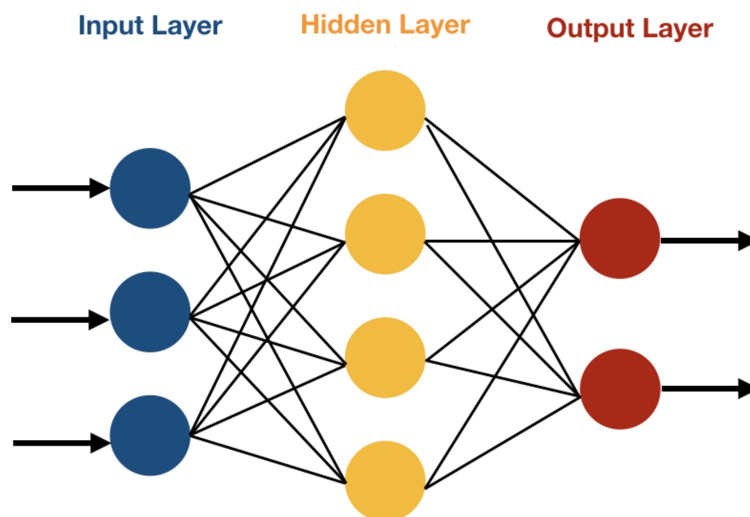


Figure 2.4: Sketch of neural networks

For example, Figure 2.5 shows the structure of determining whether the output neuron turns on via an activation function. The neural network will first give each node in the input layer a weight, and calculate through a sum function. Then, it will pass the number to an active function such as *sigmoid* to decide whether to return 1 or 0.

However, the standard neural networks do not support temporal data very well. The reason is that those networks like a feed-forward neural network are simple and do not form a cycle inside the network. For example, supposed we want to predict the

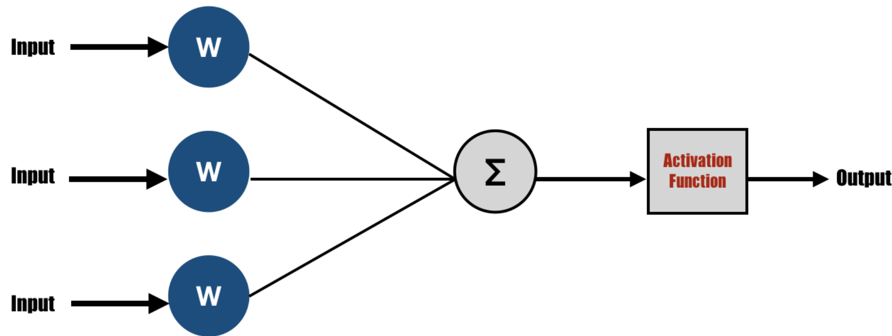


Figure 2.5: Sketch of pass through layers of neural networks

next word of "A Chinese speaks Chinese, and a Japanese speaks," which should be "Japanese." For a human being, it is easy to answer from the context. But for an old standard neural network, the eight words in the previous sentence is put into the input layer separately and ignore the relationship between each other.

Recurrent neural networks, short for RNN, is designed to solve this problem.

2.2.2 RNN: Recurrent Neural Networks

For a traditional neural network, it does not take the implicit relation between two adjacent input into consideration, as shown in Figure 2.6. The reason is that parameters in the hidden layer do not change. However, if the data is temporal such as a sentence, each word should not be separated. In other words, we could guess the next word in terms of the context.

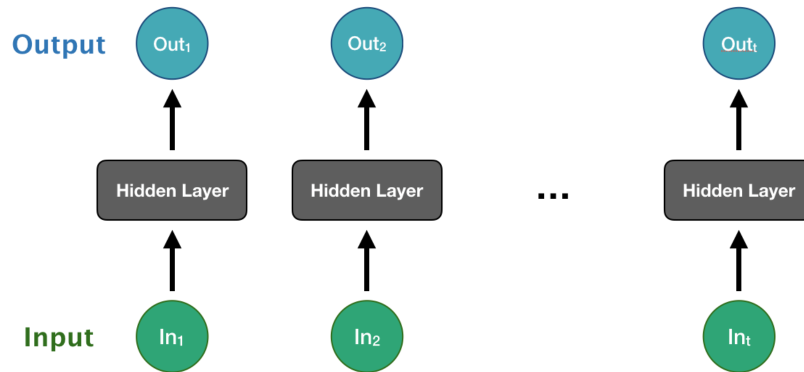


Figure 2.6: Structure of standard neural networks

In contrast, RNN modifies arguments of the current input's hidden layers considering previous hidden states, as shown in Figure 2.7. Thus, the inputs are connected by transferring these arguments through the adjacent hidden state.

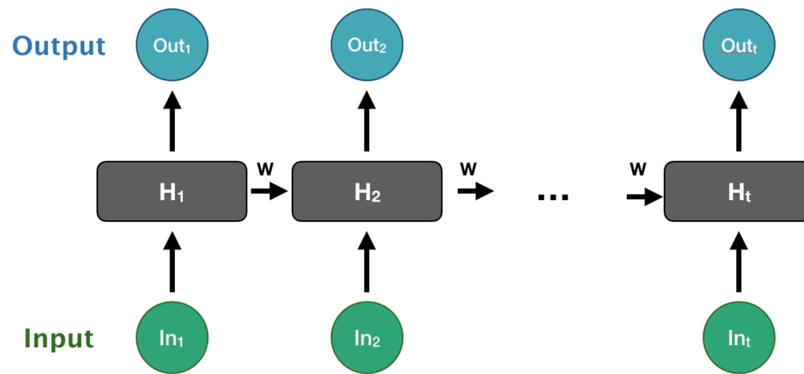


Figure 2.7: Structure of recurrent neural networks

However, a recurrent neural network can not predict precisely and accurately when the input sequence is too long. It is because the hidden layer of an input considers all previous hidden states, which leads to forgetting the initial information of the

sentence.

To be specific, recurrent neural networks use a method called backward-propagation, which would propagate the error from the last time to the beginning. And since the recurrent neural network takes all hidden states into consideration, it will cause a gradient vanishing or exploding problem.

Gradient vanishing means the error would varnish to 0, and gradient exploding means the error would explode to an infinite number. Both would make the networks forget the information from starting inputs. This makes the traditional recurrent neural networks hard to be trained.

LSTM is designed to solve this issue.

2.2.3 LSTM: Long Short Term Memory

LSTM, stands for long short term memory, is created to solve the gradient vanishing and exploding issue mentioned in the section 2.2.2.

The reason why traditional recurrent neural networks cause gradient vanishing or exploding problems is that the neural networks remember all related information even it is not important enough.

Consider a movie that has a mainline and several sidelines. Although each plot in the sideline affects the mainline more or less, the end of the film only needs the information from a few sidelines. RNN is similar to remember all the sideline plots. Thus, the audience would forget some essential plots if the movie is too long, while LSTM has a mechanism to decide whether to remember it or not.

To be more specific, LSTM imports a concept called **forget gate** that has the capacity to decide whether the information from the current timestamp should be remembered or not.

Figure 2.8 shows the mechanism of LSTM. The red line is the main line, the blue one is the side line, and the green one is the forget gate. If the input influences the mainline, then it would be added to the mainline by a weighted function. Then, the forget gate would forget some irrelevant information from the mainline. Finally, the output depends on the input information from both the mainline and the sideline.

There are several reasons why we choose to use a LSTM model.

Firstly, an LSTM model usually has a good performance on temporal data. For a program, each token is typed continuously. Therefore, LSTM is a suitable approach

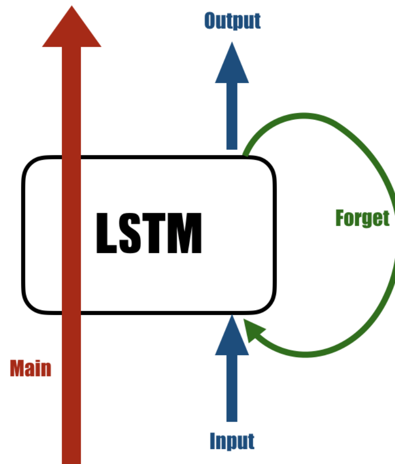


Figure 2.8: Long short term memory

to manipulate a program written in a programming language.

Secondly, we expect the LSTM model to exploit the implicit information from a program's context in order to improve the original keyword programming. An LSTM model considers the relationship between each token like what we introduced in section 2.2.2.

Third, since LSTM is a mature technique, there are a lot of implementations that are easy to use. For example, TensorFlow, Pytorch, or Keras all have their implementations that enable a user to use such a tool without specialized knowledge about machine learning.

2.2.4 Neural Text Generation

Neural text generation predicts the next likely words or sentences given some articles by using a neural network. For example, if we use all of Shakespeare's works as the training data set, we can use the model to write some sentences in a Shakespeare's style.

The prepossessing of the training data is straightforward. We need to create a bunch of pairs includes the previous words and the next word.

For example, in Figure 2.9 , we create five pairs of training data from the sentence

"This is a dog named Bark." The first one begins with the first words, which is "This" and outputs the next words "is." Similarly, the following input is the first two words, and output is the third word. And we do the same operation until the last one of the sentence.

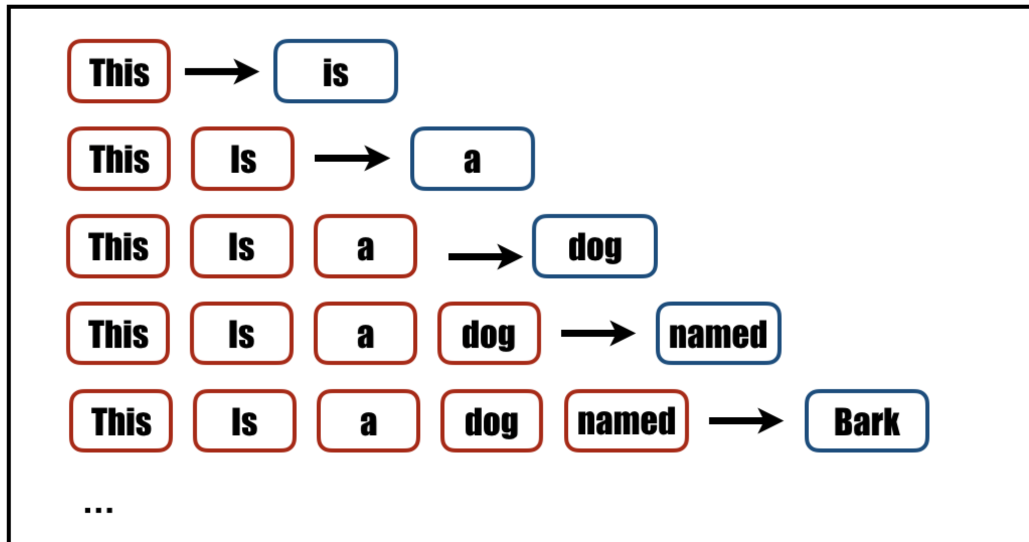


Figure 2.9: Prepossessing data of neural text generation

After training the neural networks, the model could be used to predict the next word given a sequence of words. For example, given the sentence "This is a cat" then the model would recommend the word named.

We use this idea to exploit the implicit information from the context to improve the input and generation part of the existing keyword programming. And we will discuss the details in the Chapter 4.

Chapter 3
Problem

In this chapter, we describe the problem of the existing keyword programming in details.

3.1 Difficult to generate a complicated expression

It is also difficult to generate a long-expression in the existing keyword programming system. And there are two reasons.

3.1.1 Generate duplicated expressions

First, the existing keyword programming system limit the highest height of a function to 3, because it will generate too many expressions when the height is over 3. One of the reasons that the system has abundant generations is it will repeatedly generate an expression when the depth is more than 3.

These expressions contains a method invocation expression or an instance creation expression.

Figure 3.1 shows an example given a method *concat* which receive a *String* type object and also return a *String* type, and it has a parameter which is in the type of *String*.

In the depth one, suppose we have two variable **a** and **b** where both type are *String*.

In the depth two, we can obtain four generations including *a.concat(a)*, *a.concat(b)*, *b.concat(a)*, and *b.concat(b)*.

By far, it seem no problem, but we would get a duplicated expression in `depth` 3 by using the generation expression. We found that expressions generated in `depth`

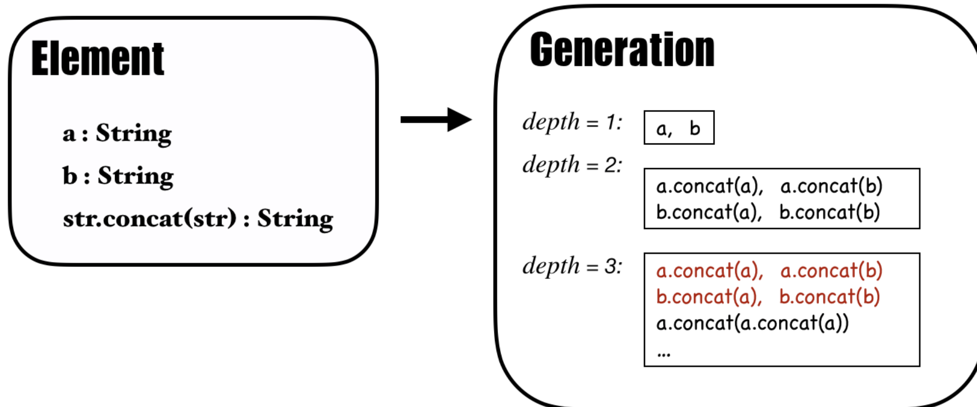


Figure 3.1: Generate example of the existing keyword programming

2 will also satisfy the condition to be generated again in `depth` 3.

Therefore, there will be a number of identical generations by using the original generation algorithm.

3.1.2 Score function limits long generation

In section 2.1.3 score function, we have described the rule for a keyword programming system to evaluate a generated expression. And the score will decrease when the depth is increased. This causes a longer expression to obtain a lower score.

Furthermore, when the expression becomes longer, it will contain more words that are not in the keyword query.

3.2 Recommendations are not likely to be used

By using the existing keyword programming to complete a program, the recommendation list always contains some expressions that are not often used.

For example, the generation with the highest score is *System.err.print(msg)* given the keyword **print msg**. Although the expression contains all of the keywords, it

actually does not match the programmer's intention.

Chapter 4
Proposal

4.1 Limit the amount of generation

The amount of generation increase exponentially.

Considering the example that we described in the section 3.1.1. If we want to generate all available expressions given two local variables `a` and `b` and a method `concat`.

It begins with two expressions in the depth one. And in `depth` two, there are four expressions. In `depth` three, there will be thirty-two expressions. And 1408 expressions in `depth` four, 2089472 expressions that are not repeated in `depth` five, and so on.

For a static programming language such as Java, we can limit the amount by the type rule. But for a dynamic programming language such as python, it seems impossible to use these kinds of systems to predict a complicated expression.

To solve this issue, we use a graph search algorithm called beam search. Beam search is a heuristic search algorithm that cuts the branch with lower priority. Figure 4.1 shows the procedure of a beam search algorithm.

For each depth, we only remain the branch with the highest score and eliminate others. And the user can set the number of how many expressions to be saved by themselves.

Although there is a possibility that the branch which generates the desired expression is cut by this approach, since a code recommendation system can not reach 100 percent accuracy, it is acceptable when this thing occurs.

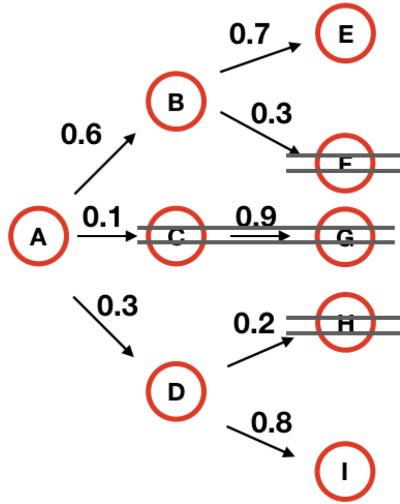


Figure 4.1: Beam search: A search algorithm

4.2 Modify the ranking algorithm

Another approach is to modify the score calculation algorithm for each expression.

We calculate the score by not only the original rules modified some weights but also another mechanism considering the probability of each token.

Figure 4.2 shows the algorithm to calculate a score according to the probability of each token.

1: **procedure** GETSCORE(context, expression)
2: $\quad$ $score \leftarrow$ GETSCOREORGINAL($expression$)
3: $\quad$ **for all** token **in** expression **do**
4: $\quad\quad$ $score \leftarrow score +$ GETPOSSIBILITY($context, token$) - penalty
5: $\quad\quad$ $context \leftarrow context + token$
6: $\quad$ **end for**
7: $\quad$ **return** $score$
8: **end procedure**

Figure 4.2: Modified ranking algorithm given previous context

It is similar to the procedure in the section 2.1.1. We separate the expression at first and get a list of tokens.

We start with calculating the probability of the first token given the previous context.

Then concatenate this token to the previous context, and use it to calculate the probability of the next token. We repeat this until the last token. And add all probabilities together.

Subsequently, we calculate the score following the previous rules. And then, the final score is the sum of these two values.

Since the user can change the weight later, we do not discuss too much about this part. Instead, we assume a programmer should modify the weight for different purposes.

For example, if the user wants to make the program seem more concisely, they could raise the penalty of generating longer expression in order to more likely to generate a shorter expression.

Chapter 5
Implementation

We build a plug-in for Eclipse and name it ACKN (Auto-Completion with Keyword programming and Neural generation) to implement our proposal.

We have three reasons why we choose Eclipse.

First, Eclipse is an integrated development environment that offers several useful tools to support a user to build a system. In this paper, with the help of ASTParser, Java Model, and Search Engine from Java Development Tool (JDT), we can achieve the necessary information to generation expressions from a source file.

Second, Eclipse is one of the most popular environments, especially for Java programmers. Therefore, it is essential to build an intelligent code recommendation system that assists hundreds of thousands of programmers to save their precious time.

At last, It is easy for a programmer to build a plug-in with the Plug-in Development Environment (PDE), which is a convenient project of Eclipse. ACKN is also built by this project and can display a recommendation list window on the editor, just like the default code completion system of Eclipse.

And same to the existing keyword programming, we divide our implementation in three phases: input, extraction, and generation. A neural generator based on LSTM is used in the input and generation section, we will describe it separately in the section 5.4.

5.1 Input

The input of ACKN is similar to the one in the previous research, which is a bunch of keywords.

5.2 Extraction

By far, we have only considered recommending expressions, including method invocation expression and field access expression. In order to generate those expressions, we need to extract all available information from each local variable, object class, field, and method.

We chose ASTParser, Java Model, and Search Engine from JDT to obtain necessary information from the editing source file to generate all possible expressions.

Moreover, since Java is a static programming language, in order to assure each generation is correct, we also need extra information, including:

- the return-type and the object class of each local variable and each field,
- the receive-type, return-type, and all parameters' type of each method,
- the current object class name.

Finally, we also need to know whether a field is static or not or whether a method is visible. Therefore, we still need to extract the information of the modifier of each field and method.

In the section 5.2.1, we explain how we extract essential information above by ASTParser, Java Model and Search Engine.

5.2.1 ASTParser

To obtain information from the context, we use ASTParser and ASTVisitor in Eclipse JDT. We would like to introduce these APIs before introducing how it works on our project.

ASTParser is used to translate a Java program to an abstract syntax tree. After parsing the whole file, we use ASTVisitor to obtain specific information, including a local variable, current class, and current package from the abstract syntax tree.

The term **abstract syntax tree**(AST) is generally understood to mean a tree structure representation of source code depends on the syntax of a certain programming language. For example, Figure 5.1 shows an abstract syntax tree for a Java code `array.addLine()`.

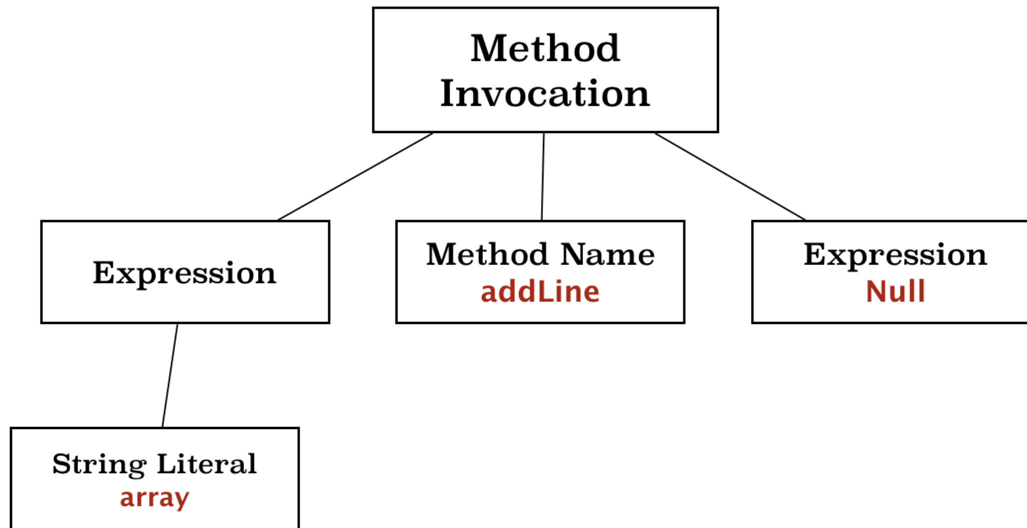


Figure 5.1: Abstract syntax tree for `array.addLine()`

In addition, the term **parse** has been applied to describe a movement that analyses the source code of a computer program and translates it to an abstract syntax tree.

Since the syntax of a programming language seldom change, and the operation to each abstract syntax tree node is often variable, it is common to use a visitor pattern to traverse the AST. In Eclipse JDT, ASTVisitor is able to visit the given node and perform some operation.

We can actually get all the information we need by parsing all the file, but the time cost would be horrible. Therefore, we only parse the current editing file and achieve the current class, current package, and the information of all available local variables.

Current Package

It is straightforward to obtain the name of the current package by visiting a PackageDeclaration node by the code:

```java
public boolean visit(PackageDeclaration node){
    this.packageName = node.getName().toString();
    return true;
}
```

Current Class

Since it is available to declare several types in a compilation unit, we need to visit the TypeDeclaration node and find one that covers the cursor position.

```java
public boolean visit(TypeDeclaration node){
    int startPos = node.getStartPosition();
    int nodeLength = node.getLength();
    if((startPos < cursorPos)&&(cursorPos < startPos + nodeLength)){
        nameOfThis = node.getName().toString();
    }
    return true;
}
```

Local Variable

It is a little bit tricky to find out all available local variables from an ASTVisitor because there is not a LocalVariable node in the syntax. Thus, we need to consider all situations and traverse each to get the set of local variables.

A variable is available in three AST node: SingleVariableDeclaraion, VariableDelcaraionExpression, and VariableDeclarationStatement. Local denotes the cursor and the variable are in the same scope.

Single Variable Declaration
A node is a single variable declaration only when it appears in a catch clause, an enhanced for-statement, or a parameter list of a method declaration.

Consider an example :
The variable *name*, *node* and *e* is a single declaration.

Variable Declaration Expression
A variable is represented by a variable declaration fragment in a variable declaration expression when it is in the initialization part of a normal for-statement.

For instance, in the program:

```java
public void function(String name) {

    for (String node : nodes) {
        try {

        } catch (Exception e) {

        }
    }

}
```

```java
public void function2() {
    for(int i = 0; i < 9; i++) {

    }
}
```

the variable *i* can be achieved as a variable declaration fragment by visiting a VariableDeclarationExpression node.

Variable Declaration Expression
Finally, We can get other variables via visiting a VariableDeclarationStatement node.

Considering the program:

```java
public void buy() {
    int gear, speed;

}
```

the variable *gear* and *speed* is the variable fragment among a variable declaration statement.

Scope
In Java, variables are always lexically(or statically) scoped, **local** means the scope where a variable can be referenced. Instead of extracting all local variable, we only take those where the cursor is in the variable's lexical scope.

Considering the program:

```java
public class Main{
    public void hoge(String name){

    }
    void fuga(int index ){

    }
}
```

If the cursor is in the line 3, then the local variable *name* can be called, while the another local variable *index* can not.

5.2.2 Java Model

Because there are a lot of files that are not public to a user, it is impossible to get all information by only parsing and visiting all source files. In addition, we use another tool from Eclipse JDT, which is called Java Model.

Java model is a tool that enables a user to manipulate a Java program. The hierarchy of a Java model is similar to a file system. Unlike the abstract syntax tree, the hierarchy of Java model is closer to the way how a Java programmer analyzes and creates a Java program, as shown below.

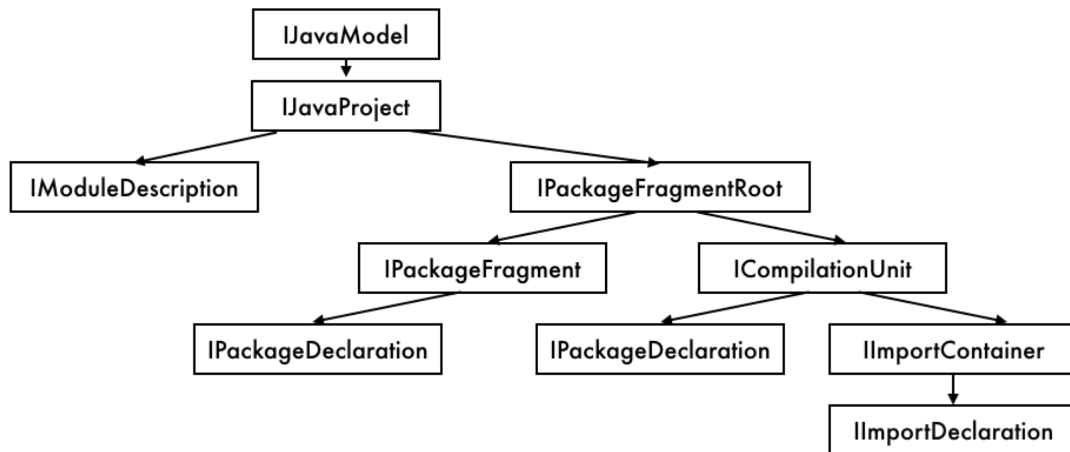


Figure 5.2: Hierarchy of Java Model

In addition, Figure 5.3 shows interfaces of Java Model which are used to manipulate the corresponding element of a Java program.

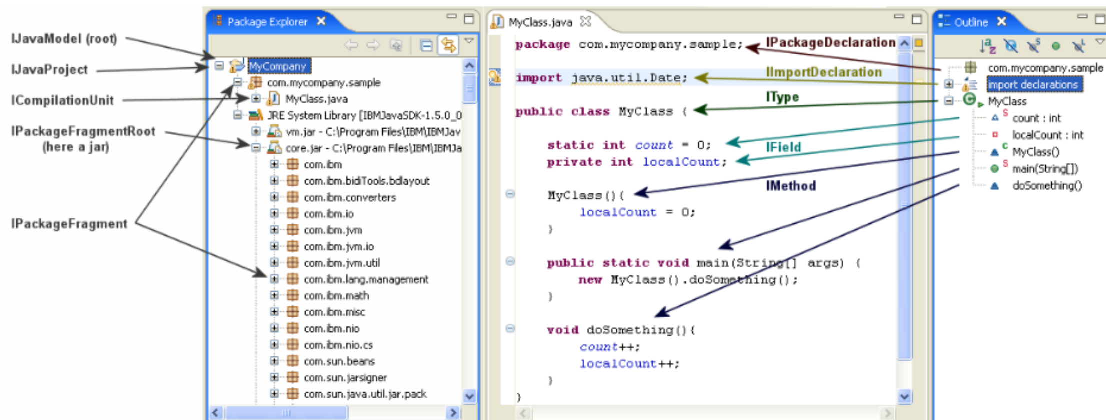


Figure 5.3: Interface for each element

For next sections, we introduce how to extract a type, a method and a field information by Java Model.

IType

In order to achieve all possible methods and fields, we should extract all possible IType at first. And there are three kinds of situations.

First, we need to extract all default types including 8 primitive types. Currently, we seem the primitive type identically as the corresponding object class. For example, *float* and *double* equal to *java.lang.Float* and *java.lang.Double*.

However, for those types in *java.lang*, we do not assume it as a default type. Therefore, the user needs to declare the class name when the program has some methods or field in that class. For example, in ACKN, the method *charAt* in *java.lang.String* is only available when the program has imported *java.lang.String*.

The reason why we do this is that the size of methods and fields in *java.lang* is too large.

Second, we require to obtain all available types from the same package. We begin with searching the **IPackageFragment** for the current compilation unit, and then traverse the path to find all possible compilation units. Then, we could get all types from each compilation unit.

The code to do this in Java is:

```
private Set<IType> getITypesFromSamePackage(){
  Set<IType> res = new HashSet<IType>();
    // get package fragment name
    IPackageFragment iPackageFragment = currentCompilationUnit.
   getParent();
    try{
      ICompilationUnit[] iCompilationUnits = iPackageFragment.
   getCompilationUnits();
        for(ICompilationUnit iCompilationUnit : ICompilationUnits){
          IType[] iTypes = iCompilationUnit.getAllTypes();
            for(IType iType: iTypes){
              res.add(iType);
            }
        }
    }
    return res;
}
```

Finally, we need to explore all available types depending on the import declaration. We only allow the user imports an object class given the fully qualified name instead of the package name. For example, if a user wants to use some methods in the *List* class, they have to use *import java.util.List* instead of *import java.util.*;*. This limitation is also used for lightening our system.

To gain all types from an import declaration name, we have to use another effective Eclipse JDT API called **Search Engine**.

Search Engine is used to search Java projects in the workspace for Java elements. Giving the name, matching rule and scope, it can provide all possible Java elements satisfied those conditions. Through this useful tool, we can extract all available types in the import declarations.

For example, if we want to get all types' name in the package *java.util*, we should provide the Search Engine.

1. the package name which is *java.util* in this case
2. the matching rule which is exactly matched
3. a scope such as the whole work space.

IField

After we obtain all available types shaped in IType, we continue to use the Java Model to get the information of fields. We can get the modifier, name, and return type with the Java Model API.

IMethod

The way to get all IMethod elements is similar to the one in the previous section, except a method also need the set of parameter types.

5.3 Generation

Instead of using the complicated syntax of Java, we define a simple one that only applies to generate expressions. We currently do not consider the control flow, such as an if-else expression or a loop. The syntax is shown in the appendix **??**

Although it is available to support more syntax rule, it will take a long time to generate all possible expression. Thus, in our paper, we only allow a user to complete a field access expression and a method invocation expression.

We have attached the syntax rule of our system in the appendix.

In this chapter, **depth** is used to refer to the height of an abstract syntax tree of an expression. We will introduce our generation procedure divided by different depths.

Moreover, in order to shorten the generating time, we build two tables that dynamically store the available expressions for each type and depth. We named **tableExact** for the table containing expressions in an exact depth, and **tableUnder** for the table that has expressions under each depth.

Generate expressions in the depth one

In `depth` one, the system has the capacity to generate a variable or an instance of an object.

It is pretty straightforward to implement this part. For each accessible type, we create an instance and extract local variables that return the type.

For instance, in the program

```
 1  package example;
 2
 3
 4  public class Test {
 5⊖     public Test() {
 6
 7     }
 8⊖     public void foo(int index) {
 9         int number;
10         |
11     }   index: int   Score : -0.059
12  }      number: int   Score : -0.059
13         new Test(): Test   Score : -0.23
14
15
```

Since the cursor is in line 10, there are two local variables including *index* and *number*. Besides, the system could also generate the initial constructor of the object class *Test*.

Generate expressions in depth two

It becomes a little bit complicated in `depth` of two, because we need to enable the system to generate a method invocation or a field access expression.

In this section, the system can generate 3 kinds of expressions in `depth` 2, which are instance creation, field access, and method invocation expression.

First, we define an instance creation in `depth` two when all of the parameters are in the depth of one. For example, in the program,

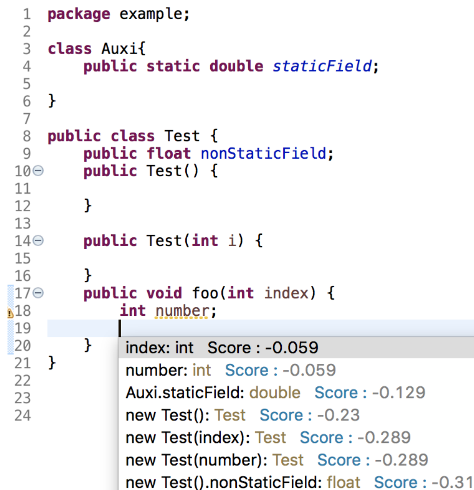
```
 1  package example;
 2
 3
 4  public class Test {
 5⊖     public Test() {
 6
 7     }
 8
 9⊖     public Test(int i) {
10
11     }
12⊖     public void foo(int index) {
13         int number;
14
15     }   index: int   Score : -0.059
16  }      number: int   Score : -0.059
17         new Test(): Test   Score : -0.23
18         new Test(index): Test   Score : -0.289
19         new Test(number): Test   Score : -0.289
```

Since only three variables are in the depth one, and *index* and *number* are available regarding the cursor position, only the fourth expression *new Test(index)* and fifth

expression *new Test(number)* are two instance creations in `depth` two.

Secondly, we define a field access expression is in `depth` two if the field is static or the object expression is exactly in `depth` one.

For instance, in the program



the third recommendation *Auxi.staticField* is a static field access expression in `depth` two. And the seventh recommendation *new Test( ).nonStaticField* is a non-static field access expression where the object expression is an instance creation in `depth` one.

Finally, a method whether static or not is in `depth` two when each parameter consists of expressions in `depth` one. And a type is assumed as `depth` one through this paper.

Since the number of parameters of a constructor or a method is arbitrary, we use a recursive algorithm to generate all possible expressions rather than a nested loop.

The algorithm to get all instance creation is:

Notice that we use a bottom-up way to generate an expression with arbitrary parameters. For example, for a constructor *new Test(int first, int second)*, the system will initially generate all possible expressions for the second parameter and then determines what appears in the first parameter position.

```
procedure GenerateInstanceCreation(paraNum, subExps, paraTypes)
    if paraNum == 0 then
        res ← GenerateWithSubExps(subExps)
    else
        paraType ← paraTypes[paraNum − 1]
        paraExps ← GetExpressionFromExactDepthOne(paraType)
        for all expression ∈ paraExps do
            subExps[paraNum-1] ← expression
            GenerateInstanceCreation(paraNum-1, subExps, paraTypes)
        end for
    end if
end procedure
```

Generate expressions greater than depth two

For a field access expression, the generation mechanism is similar to the one in the previous section. Nevertheless, we use a bit computation algorithm to prevent generating duplicated expressions.

In the section 3.1.1, we mentioned that the previous generation algorithm would generate an expression repeatedly. And in the section proposal, we find out a solution is to give a limitation to generate expressions with multiple parameters. The rule is that at least one of the parameter is exactly in **d-1** when generating an expression in depth **d**.

For example, if we want to generate all possible method invocation in `depth` 3 given a method *add(paraOne, paraTwo)*. There are three situations that satisfy the limitation. Either `paraOne` or `paraTwo` is exactly in `depth` 2, and the other is in `depth` 1. Otherwise, both of them are in `depth` 2.

We use a binary representation where **1** represents expressions in exactly **d-1** and **0** represents expressions in `depth` under **d-2**. As a result, the three situations for the former example *add(paraOne, paraTwo)* can be represented as (**0**, **1**), (**1**, **0**), and (**1**, **1**).

5.4 Neural Generation

The neural generation of ACKN has two parts. One is to generate a model by using LSTM. And another is a program that is able to predict the probability of an expression given the context.

5.4.1 LSTM model

Prepossessing

Before we train the neural networks model, we have to process our training data to make it perform better. In our research, we have to approach to increase the quality of our data.

First, we eliminate all comments of each program, since keyword programming does not take the comment information into consideration. In addition, the comment would influence the training and be likely to generate a sequence of tokens that is similar to a natural language.

Second, because it takes a long time to train an LSTM model, we use a word2vec technique to decrease the dimension of each word to shorten the training time.

Training

In our research, we use the Keras LSTM as our neural networks implementation. And the activation function is *Softmax*, the optimizer uses the Adam optimization algorithm, and we use *sparse categorical cross entropy* function as our loss function .

5.4.2 Connect by Socket

Since we currently implement the keyword programming in Java and neural generation in python, we need a way to pass the information through these two systems.

We choose the socket programming, which seems the Java program as a client and the python program as a server.

First, the Java program will send the context information to the python program and get a sequence of characters that represent next likely tokens calculated by the neural generator. Then, at the generation step, the Java program will send each generated expression to the server and get a number denotes the probability of the expression.

Chapter 6
Evaluation

6.1 Tasks

We prepare 15 tasks to evaluate ACKN. For each task, it includes an expression, a program with a hole, and a keyword query.

First, we decide the expression by ourselves such as `System.out.println(result)`. Then, we find 21 different programs from Github that contains the code of the expression. We add 20 of them to the training set. For the remaining one, we erase the code of the expression, and create a test program with a hole. Subsequently, we make a keyword query that can represent the meaning of the expression such as **print result**. Then we run the program on ACKN given the keyword query, and compare the rank between ACKN and the original keyword programming system.

Table 6.1 shows the keyword query and the expected expression that we made. And Table 6.2 shows the rank of the expression and the top recommendation by the original keyword programming system. Table6.3 shows the rank and the top recommendation by ACKN.

6.2 Result

While the training set is not big enough, we can see that:

- Among 10 of 15 tasks, the desired expression are on the recommendation list
- Among 6 of 15 tasks, the desired expression are in the Top-5 of the list.

No.	Keyword Query	Expected Expression
1	**print result**	*System.out.println(result)*
2	**print error**	*System.err.println(error)*
3	**print result**	*System.out.print(result)*
4	**print error**	*System.err.print(error)*
5	**get time**	*System.nanotime()*
6	**get time**	*System.currentTimeMillis()*
7	**str at index**	*str.charAt(index)*
8	**upcase str**	*str.toUpperCase()*
9	**lower str**	*str.toLowerCase()*
10	**str from begin to end**	*str.substring(begin, end)*
11	**limit capacity to min**	*sb.ensureCapacity(min)*
12	**get address of host**	*InetAddress.getLocalHost()*
13	**input**	*new Scanner(System.in)*
14	**read standard in**	*new BufferedReader(new InputStreamReader (System.in))*
15	**load resource name**	*Thread.currentThread().getContextClassLoader() .getResource(name)*

Table 6.1: keywords and expected expressions of 15 tasks

6.3 Discussion

6.3.1 Method v.s. Variable

Since local variables are named differently by each programmer, it is difficult to predict a likely variables name by neural networks. An available approach is to raise the weight of a local variable. For example, we could add 0.1 to the score when a token is the local variable.

In contrast, the neural generator performs well in predicting a method name.

6.3.2 Speed

Currently, it takes a long time to predict expression once. And there are two reasons.

First, even we use beam search to limit the number of generations, there are still many expressions when the depth is over 4.

Second, we use python to write the neural generation program, and Java to write the keyword programming system. It also costs time to communicate with each other.

Task	Position	Top expressions
1	×	*System.err.print(result)*
2	×	*new PrintStream(error)*
3	2nd	*System.err.print(result)*
4	3rd	*new PrintStream(error)*
5	2nd	*System.getProperties()*
6	4th	*System.getProperties()*
7	1st	*str.charAt(index)*
8	10th	*str*
9	1st	*str.toLowerCase()*
10	4th	*str.substring(end, begin).toString( )*
11	×	*sb.append(min).insert(sb.capacity(),sb.toString())*
12	2nd	*local.getHostAddress()*
13	24th	*new Main().readInputUntilEndOfLine()*
14	29th	*new InputStreamReader(System.in).read()*
15	×	*ClassLoader.getSystemClassLoader() .loadClass(ClassLoader .getSystemResource(name).getRef())*

Table 6.2: position and top result by using the original keyword programming

Alternatively, we could use an LSTM model written in Java, such as Deeplearning4j.

6.3.3 Accuracy&Precision

Although we could get the desired expression in 10 tasks by ACKN, the data set is not big enough to use it into practice. We could collect projects on GitHub, which have stars more than five.

In addition, we could also improve the LSTM model to increase the accuracy of prediction. For example, we can add another LSTM layer which manipulates a file from the tail, or we could add the attention mechanism to the model.

The desired expression is in the Top-5 on the recommendation list in the case that ACKN could recommend successfully. Therefore, we believe that it is available to increase the precision of keyword programming by concerning the context.

6.3.4 Input priority

In this paper, we focus on the problem of distinguishing *System.err.println(f.getName())* and *System.out.println(f.getName())* given **print name of f**. But the weight for each

Task	Position	Top expressions
1	5th	*System.err.print(result)*
2	×	*new PrintStream(error).println()*
3	2nd	*System.err.print(result)*
4	×	*new PrintStream(error)*
5	1st	*System.nanoTime()*
6	1st	*System.currentTimeMillis()*
7	1st	*str.charAt(index)*
8	9th	*str*
9	1st	*str.toLowerCase()*
10	×	*str.substring(end, begin).toString( )*
11	30th	*sb.append(min).insert(sb.capacity(),sb.toString())*
12	25th	*new Main().getLocalHost().isMulticastAddress()*
13	22th	*new Main().readInputUntilEndOfLine()*
14	×	*new InputStreamReader(System.in,input.readLine())*
15	×	*ClassLoader.getSystemClassLoader()* *.loadClass(ClassLoader* *.getSystemResource(name).getRef())*

Table 6.3: position and top result by using the AKCN

keyword should be different, which means the weight for **print** should be higher than **of**.

Since keyword programming is similar to natural language processing, and we can get inspired by the skills of natural language processing.

One of the famous weighting skill is term frequency-inverse document frequency, short for TF-IDF. This method calculates the weight of a word by multiplying the term frequency in a document and the inverse document frequency in a bunch of files. If we want to get the weight of each keyword, we could use questions on Stack Overflow and calculate the TF-IDF value.

Chapter 7
Related Work

This work is based on the technique proposed in Little and Miller, 2009, where recommends expressions given a keyword query.Campbell and Treude, 2017, and Nguyen et al., 2016 also input a keyword query. But their goal is to translate the keyword query to code snippets on Stack Overflow.

Bruch, Monperrus and Mezini, 2009 seeks for the programs that the order of method invocations are similar to the editing file. And they also build an Eclipse plug-in named CodeRecommenders.

Robbes and Lanza, 2010 analyses the programming history of a programmer, and is more likely to recommend an expression that is recently used. One of the drawbacks of these systems is that the user can never get the desired expression if they have not used it.

Liu et al., 2016 and Hidehiko, n.d. also use an LSTM model on code completion. However, they manipulate the source file in the form of the abstract syntax tree and predict the next node to complete the whole system.

Chapter 8
Conclusion

In this paper, we propose a keyword-based code recommendation system improved by concerning the context information.

We use an LSTM model that can predict the probability of the next token given the context.

We modify the ranking algorithm of the existing keyword programming, which considers the probability of each token.

Furthermore, we use a graph search algorithm called beam search to limit the size of generation and get multiple recommendations.

Finally, we build a plug-on on Eclipse and name it ACKN.

We testify ACKN by recommending fifteen expressions that the original keyword programming system is hard to predict. 10 of 15 can show on the representation list, which contains 30 candidates. And 6 of them can be shown as the Top-5 of the recommendation list.

The result of this study indicates that a neural generator can predict the next method invocation expression that would be used when there is no local variable inside it.

As a consequence, we believe that our method could be used in predicting a method invocation that does not rely on any variable.

References

Bruch, Marcel, 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*. ACM, pp. 213–222 (cit. on pp. 6, 47).

Campbell, Brock Angus and Christoph Treude (2017). 'NLP2Code: Code snippet content assist via natural language tasks'. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 628–632 (cit. on pp. 4, 5, 47).

Han, Sangmok, David R Wallace and Robert C Miller (2009). 'Code completion from abbreviated input'. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, pp. 332–343 (cit. on p. 5).

Hidehiko, Li Dongfang Masuhara (n.d.). 'ASTToken2Vec: An Embedding Method for Neural Code Completion'. In: () (cit. on p. 47).

Hu, Sheng et al. (2019). 'Autocompletion for Prefix-Abbreviated Input'. In: *Proceedings of the 2019 International Conference on Management of Data*. ACM, pp. 211–228 (cit. on p. 5).

Little, Greg and Robert C Miller (2009). 'Keyword programming in Java'. In: *Automated Software Engineering* 16.1, p. 37 (cit. on pp. 6, 7, 47).

Liu, Chang et al. (2016). 'Neural code completion'. In: (cit. on p. 47).

Nguyen, Thanh et al. (2016). 'T2API: synthesizing API code usage templates from English texts with statistical translation'. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 1013–1017 (cit. on p. 47).

Oka, Akio et al. (2017). 'Live Data Structure Programming'. In: *Companion to the first International Conference on the Art, Science and Engineering of Programming.* ACM, p. 26 (cit. on p. 1).

Robbes, Romain and Michele Lanza (2008). 'How program history can improve code completion'. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, pp. 317–326 (cit. on p. 6).

– (2010). 'Improving code completion with program history'. In: *Automated Software Engineering* 17.2, pp. 181–212 (cit. on p. 47).