A Syntax-driven Approach for Natural Language to Programming Language Translation to Realizing Literate Programming in Java

by

Hung Phan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan , Major Professor
Mary Jones
Bjork Petersen

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this thesis to my family. Without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and financial assistance during the writing of this work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. August Tanner and Dr. Lewis Hargrave. I would additionally like to thank Dr. Tanner for his guidance throughout the initial stages of my graduate career and Dr. Hargrave for his inspirational teaching style.

ABSTRACT

Computer Programming should be considered as art . Literate Programming (LP) (Knuth (1984)) is a programming paradigm proposed by Donald Knuth to realize this idea. LP helps developers to make the source code to be more like a literature, by the ability of integrate the explanation in natural language (NL) along with original source code. However, since its appearance in 1984, LP has not been used as a popular programming paradigm. One of the challenges is that developers need to write the corresponding source code manually for any NL parts they defined to make the program compilable. In this project, we proposed Natural Language to Programming Language (NLPL) system, to reduce the effort of writing code. Our NLPL system allows developers to write explaination as code comment in NL and automatically generating the respected source code following the requirements of the NL parts. The code generation is done by the NLVisitor module we developed for providing translation rules at Natural Language syntax level and alleviating indirect references problems between languages. The experiment on the set of 52 comments from the code suggestion tool AnyCode (Gvero and Kuncak (2015)) achieves the result at 75% in top-1 accuracy, which outperforms this prior work (62%).

CHAPTER 1. OVERVIEW

1.1 Literate Programming

The notion of Literate Programming (LP) [Knuth (1984)] provides a potential programming paradigm which brings advantages to software developer. Considering that writing source code need to be seen as an art work, LP handles the input source code by two parts: documentation and implementation. The implementation parts contain the source code in a specific programming language like Java or C/C++. Unlike original programming language (PL) paradigm, the full implementation defined in a LP source file can be split in several parts, which are interleaved by documentation parts. These parts provide descriptions about expected code in natural language. Each documentation parts are corresponded with an implementation part. A pair of corresponding documentation and implementation part has the same title's prefix.

An example of a LP file, called literate file, is shown in Figure 1.1. This file, which is written by C++ published from [Knuth (2009)] shows the destructor of the XString class. The implementation part of this method is defined in other area and it is aliased by a documentation part started with "Decrement". The documentation part contains the description of its corresponding code, including what variable needs to be decreed.

```
127 @ The destructor also has to worry about the reference count:
128 @<|Xstring|...@>=
129 Xstring::~Xstring()
130 {
131   @<Decrement reference count, and remove |p| if necessary@>;
132 }
133
134 @ @<Decrement...@>=
135 if (--p->n == 0) {
136   delete[] p->s;
137   delete p;
138 }
```

Figure 1.1 Example of XString.w Literate Programming file in C++ from [Knuth (2009)]

LP provide advantages to software developers. This programming paradigm allows users to understand source code better. It provide a layer to help developers able to read both of natural language (NL) representation and programming language representation of the code. In example showed in Figure 1.1, developers know what action need to do for the destructor in the documentation part, which is decrementing value of a variable, and how to do the action by a decrement assignment and an if statement. Besides, LP helps developer read the documentation parts aligned with the programming parts, which they can check the explanation of behaviors of each program elements instead of reading the implementation and documentation separately. This way of representation make developers be able to imagine about how the source code work for each implementation parts and be able to debug the code easier. In overall, LP provides the more understandable form of source code, including what does the code do and how it works by descriptions in NL and descriptions in PL.

1.2 Challenges of realizing Literate Programming

Realizing LP requires both representation of documentation and implementation parts for each source code file. There are two possible trends to derive these parts, however they faces the problems of effort consuming and low accuracy.

1.2.1 Manually writing documentation and implementation

Since the appearance in 1984, LP has not been applied as a popular programming language [StackExchange (2010)] This fact is surprising since LP provides advantages for code understanding. According to]cite, the main reason is that manually writing the code given documentation parts and manually documented the code given implementation parts are expensive tasks, especially with people who only have background in writing documentation or only have background in writing the code.

For people who are focusing their work on writing software documentation, doing the implementation part of the documentation are usually infeasible task for them to complete. They know

about what elements need to be defined or handled in the implementation and know how program elements involved implementation parts. However, their knowledge is represented by natural language. The most straight forward way for them is learning about programming language to have background of writing code. However, it is not a cheap stacks since it requires 4 years on average for training a Computer Science undergraduate student in the US. NL and PL has many differences how they abstracting the software explanation. For example, in natural language, people can use some indirect reference like pronoun such as "it", "they" while their is no such pronoun representation in some common programming languages like Java. Besides, people who are writing software requirements might need to work with projects in several languages instead of a single programming language, cause it is impossible for them to infer the implementation parts even if they already have background in one language.

From vice versa, people who have experiences in doing the implementation parts of the code might have more background for manually writing documentation parts of the code they write. This could be possible and recommended for any developers who are students to do the documents correspond to their small software projects, to make them understand the code better and check bugs in the code faster if the bugs exist. However, in industrial projects which contains more than thousands line of codes, the cost for writing documentation related to each parts of code are expensive, while it can slow down the development process of a software project as half, since developers need to write the explanation for the code they write. In addition, the code does not stay the same and code are updated days by days following requirements of users. This fact brought challenges for software developers to manually write documentation although they have good background for explaining the code in natural language. In overall, manually writing both documentation parts and implementation are expensive. So that, approach for automatically deriving documentation and implementation are important.

1.2.2 Automatically inferring implementation by Machine Translation

In translation between natural languages, phrase based machine translation (PBMT) and Neural Machine Translation (NMT) are two common machine translation (MT) approaches that show their effectiveness for the inference problems. This fact brings MT as the most used translation engines [Young et al. (2018)]. We might think about a solution for the software engineering is that applying MT to inferring implementation from documentation, given the intuition that both implementation and documentation have the same intension of describing how computer program works. However, researches on using MT in this area still not have good results yet [Barone and Sennrich (2017)], due to several following problems.

1.2.2.1 Lack of high quality Parallel corpus

In natural language processing, current MT techniques rely on large scale corpus, which contains 10000 to millions pairs between different languages [Luong et al. (2017)]. Machine learning algorithms like Bayesian learning are used to learn the mapping between source and target languages. The corpus needed to be prepared as pairs of sentences, and the target languages need to be written manually by linguistic experts. However, in software engineering, building such a parallel corpus for documentation and implementation manually are expensive. Another approach is to relying on documentation written as natural language code comments. This seems to be a good direction since large scale code corpus contains millions line of code (LOC) and there are textual description in the form of code comments for each source code files. However, this type of documentation doesn't guarantee to have the same intent of describing the behavior of related source code. For example. this documentation contains "TODO" tags which are automatically generated by the IDE, or contains comments that are actually source code which is commented by developers. In fact, such a corpus generated by documentation in the form of code comments and implementation contains noise and cause very low quality inference in MT models [Barone and Sennrich (2017)] . Along with the problems from the shortage of good parallel corpus, MT also facing the problems from the differences between natural language and programming language. [Pulido-Prieto

and Juárez-Martínez (2017)] provided a survey that summarized the main differences between two languages which hindered an MT models for learning mapping between two languages. In general, it comes from different purposes of translation between natural language processing (NLP) and program generation, and more important, natural language and programming language have different mechanism of indirect reference (IR) management mechanism [Pulido-Prieto and Juárez-Martínez (2017)].

1.2.2.2 MT for Natural Language to Natural Language: Descriptive Model

In NLP, the main purpose for inferring the target language is to helping user understand about the main idea of the source sentence. In the other words, it solved like providing the description for the content of source language, which usually include what is the subject, what are the verbs and what are objects used in the sentence in a form of a sample subject-verb-object (SVO)structure sentence. Let's see an example in Figure). In this example, which we observed from NMT system for Germany to English translation published by Tensorflow [Luong et al. (2017)], the generated neural machine translation result shows almost the same with the expected sentence except one word "Halef" which is a last name. From this generated sentence, users can easily understand the meaning of the sentence and this kind of missing words error are acceptable. In NL, MT worked as a descriptive model, which allows the translated result missed one or two words compared to the expected sentence while users still understand.

Source (German)	Hafez Abu Saeda ist wü tend über diesen for ci erten Verfassungs prozess.
Expected (English)	Hafez Abu Saeda is furious about this forced constitutive process.
Result (English)	Abu Saeda is angry at this forced constitutional process.

Figure 1.2 Example of translated result produced by Neural Machine Translation in German-to-English translation engine

1.2.2.3 MT for Natural Language to Programming Language: Generative Model

In contrast with NL-to-NL translation, the direction of natural language to programming language requires a very precise generated result. We can look at an example in Fgure 1.3. SpecTrans [Phan et al. (2017)] is a statistical machine translation system for inferring between documentation and implementation for Java specification. The phase of inferring implementation returns 27 % of syntactically correct. SpecTrans returns 38 % of results are close to the expected result like this example. In the example, although we only used two edit actions to get the correct result, the translated implementation are still syntactically incorrect and cannot use directly as an generated source code. The problem of syntactically incorrect doesn't appear only in documentation to implementation translation but also in translating between object oriented programming languages. SemSMT [Nguyen et al. (2013)] shows that there are over 50% of translated results are syntactically incorrect. In overall, generating code from source, even if documentation or code from other programming language is an error-prone task by machine translation.

Source (Doc)	throws IndexOutOfBoundException if index is less than zero
Expected (Impl)	if (**fromIndex** < 0) throw new IndexOutOfBoundsException ( ) ;
Result (Impl)	if (**index** < 0) throw new IndexOutOfBoundsException ( ) **!= null)**

Figure 1.3 Example of translated result produced by SpecTrans for Documebtation-to-Implementation translation

1.2.2.4 Differences in indirect references between Natural Language and Programming Language

[Pulido-Prieto and Juárez-Martínez (2017)] summarized problems of the generation of code from natural language of several existing tools. In general, the most important is that they have unique ways for handling indirect references. An example of such types of differences is shown by example 1.4. In this example, a variable x can be described by the pronoun "its" in natural language as an indirect reference to variable x. However, in the implementation, there is only

one way for represent variable x. These differences brings challenges for MT system to learn the mapping between languages. We will describe about types of indirect references between natural language and programming language in the next section.

```
2 public class ExThesis1 {
3⊖    public static void main(String[] args) {
4 //      initialize x and set its value to zero
5         int x=0;
6     }
7 }
```

Figure 1.4 Example of description for initialization in Java

1.3 Direction of a syntax-based approach for Natural Language to Programming Language translation

Realizing Literate Programming can take benefits from a module for automatically generating code from documentation. However, the building of such a module based on machine translation engines used in natural language processing faced challenges. In this project, we analyze the problem from another aspect instead of relying on statistical approach. Our intuition is that, natural language description can be represented as a natural language parser by [Socher et al. (2013)], and each typed of nodes in the parsed tree has rules for the translation by a syntax-based approach. To achieve this task, we conduct my research in the following steps. First, according to survey [Pulido-Prieto and Juárez-Martínez (2017)], we studied main problems of indirect references between two languages. Next, we studied each node types in Natural Language parser and propose rules for translation. After that, we implement a module that embedded these rules in an Natural Language Visitor based on the idea of ASTVisitor in programming language, to translate the code comment for each source files of a Java project. We build a tool as an online IDE for users to do literate programming and publish on the site[Phan (2018)].

CHAPTER 2. REVIEW OF LITERATURE

In this chapter, we will overview main differences between NL versus PL and understand meaning of each natural language elements. To get this summarization of problems between two languages, we rely on problems between NL and PL listed in) and problems we found.

2.1 Differences between Natural Language and Programming Language

According to), programming language has limit language expressiveness while natural language description has stronger expressiveness. In natural language, there are linguistic elements that have a different level of expressiveness compare to programming language.

2.1.1 Deixis

According to Oxford English Dictionary, "deixis" means "the function or use of deictic words or expressions whose meaning depends on where, when, or by whom they are used". In other words,this concept highlight that in natural language, there are words that their meaning depend on the context. Deixis are used to express the sentence in shorter way without losing its meaning. In the direction of NL to PL, along with the context of NL, we have the context from the surrounding source code.

Let's look at an example of deixis in this code snippet. In this example, "a file" here has the reference to the type "java.io.File". In this context, although the description mentioned only about a class name. There are several ways for describe an object as "deixis" like using pronouns such as "it" or "they". In the other words, deixis provides a type of indirect reference between object in natural language and object in implementation.

Another example is in figure). In this example, the verb "create" will be understood as an constructor of the DataInputStream object, with the respected argument as a file name mentioned in the NL description. This case shows another example of indirect reference, which map a verb to a constructor statement in Java. There are multi types of deixis between NL and PL which hindered automatically approach like NMT to study this mapping based on the problem of indirect references.

2.1.2 Expressiveness

In Oxford Dictionary, expressiveness means "showing or able to show your thoughts and feelings". In NL, expressiveness helps users to describe their ideas in many ways. This is due to the fact that English can cover a large set of lexical rules, grammatical rules and textual rules.) shows that there are different set of rules between different subset language of English. In the area of NL to PL, natural languages have implicit references for expressing ideas while programming language used a very explicit references. For example, we can easily see natural language description contains "previous", "next" but in the corresponding implementation, we can only have number and variables.

NL and PL have a very separate mechanism of managing the expressiveness in description related to control structures. An example is shown in figure). In this example, the description mentioned about if and otherwise case about a boolean condition. However, in the implementation, the code for the implementation is the if-then-else structure. The order of description of statement of else branch is before the term "otherwise", while "else" keyword should be before this statement in the implementation. Along with basic control structure, we can face the problem of expressiveness in many other structure and algorithms, like loop and recursion. The different of expressiveness causes the mismatch between learning the control structure of programming language given the natural language description.

2.1.3 Phrases

Natural Language description consists of one or more phrases. In NL to PL, each phrase may have a specific semantic meaning related to variables and statements. Phrases can be very complex and can be ambiguous about their mapping in the PL. Several phrases can be mapped to one statement in the implementation, while several sentences can be mapped to single control structure.

In example 1 figure), we can see that two phrases of the description "create bit set and set its 5th element to true" corresponded to one method invocation, in which the first phrase is the receiver while the second phrase mentioned about the method identifier and expected method argument. The differences in phrase translation rules can be shown on example 2. In this example, two sentences in the documentation actually mentioned about If branch and else branch of an if statement. In general, the mapping of phrase to statement in NL to PL is not one one mapping. The mapping rules between phrase in NL and statement in PL might depends on types of phrase along with context of description.

2.1.4 Anaphoric Relations

In natural languages, abstraction of types are not only in the form of class name like example we show in deixis but also in the form of providing description relate to class properties and methods. An example of anaphoric relations is shown in example). In the NL description, a noun phrase "dog" actually refers to a class name and the verb "is barking" mentioned about method bark() defined inside a Dog class. This description implies about a validation for NL description, in which ensure that bark() has to be a method of Dog object. Depending on paradigm of programming language, the corresponding implementation may be differed. For example, in object-oriented language, we must assured that the project contains this description has definition of Dog class with bark() method, while in functional language like C or Pascal, we valid our program by finding a function bask() that accept a struct "Dog".

Along with deixis, anaphoric relations provides the full support for indirect reference between NL and PL. While deixis relate to how to mention about a type information in form of class name only or pronoun, anaphoric relations focuses on expression every properties of an instance, including its methods and fields. A common of type of this relation is relation between verb to method invocation. When you see a verb in NL description along with a class name like figure). it has the potential that the corresponding implementation is about a method invocation of the class name in NL description.

2.1.5 Context

To realizing literate programming, context is an important element deciding the correctness of natural language to programming language translation. Unlike NL-to-NL translation which considered the context such as surrounding phrases or surrounding sentences given an input, the NL-to-PL considered both the natural language side and programming language side as context of given NL description.

Given a NL description, its related implementation might depend on surrounding NL description. In the example in figure). the phrase "set its 5th element to true" has a translated result as a method invocation "set" of java.util.BitSet object. We know about the receiver of this implementation based on the previous phrase "create bit set", which provides us information about a BitSet object. If we don't have the first phrase, we cannot inferred to the correct set method invocation, though we have method name along with parameters.

Along with context from NL part, information from programming language part brings an essential context for NL description. Let's see example 2 from figure. In here, unlike example 1, we don't have information about the constructor of Bit Set. However, we can predict about the receiver of "set" method identifier based on the surrounding code, in which we have the information about a BitSet object. Context of PL parts provides all possible variables and class name are valid to use

for the NL description. Context at PL level helps to restrict the implementation parts only relate to APIs that define in the scope of project, including its imported jar file and source code files. In our knowledge, current techniques for generating code from natural language considered the NL context as a full context and skip the information about surrounding code like AnyCode . In AnyCode, each NL description solved as a query for code instead of a comment, which hinders its ability for realizing literate programming, which requires NL descriptions consistent with surrounding NL context and PL context.

2.1.6 Ambiguity

Ambiguity in NL to PL translation means that one element in NL may has many interpretations, which the correct one may be varied based on the context. We can manually check to find the correct interpretation but an automatic translation engine doesn't have the ability to do that. To allow reasoning and de-ambiguous the translation result, we need to study types of ambiguity and design an algorithm for reasoning. In our knowledge, their are 2 important types of ambiguity in NL side and PL sides.

In NL side, noun and verb are two popular elements that have ambiguous in translated results. With noun, class name in NL might have more than one type annotation in PL. Information of correct type annotations can depend on the context of source file projects. In example show in figure), the class name "URL" has several type annotations, in which the most popular one is "java.net.URL", With verb, the ambiguity can happen between statement trigger or method invocation. In example 1 show on figure), the "set" verb is a assignment statement in the implementation, while in example 2 it is a method invocation set of Hash Map. There are several information from context can differentiate between two cases. In example one, the verb is used with a variable, a preposition "to" and a value that have the same type with a variable. In example two, the verb is used with two variable written consecutively. We can rely on such information to make correct reasoning.

In PL side, ambiguous can happen between translating verb as method invocation. One verb can be translated to multiple invocations. In the example show in figure), there are 12 method

APIs named set in libraries of JDK and apache common io. They have the same identifiers but their behavior are differed from each other, based on their receiver and argument. To reason the correct API for verbs, relying on the NL context including variables and value defined in the NL description might be a good direction. In overall, ambiguous is one of the most challenges for NL-to-PL translation, which make code query system like) cannot be used for literate programming, since it doesn't use enough context in PL for reasoning the correct interpretation.

2.2 Elements of a Natural Language parser

Stanford NLP) is one of the most popular Natural Language Processing toolkits. The core of Stanford NLP is based on its Stanford Parser. Stanford parser is a natural language parser, which analyses the grammatical structure of sentences and identifying type of words, such as verb, subject or object. Stanford Parser uses probabilistic parser, which uses information from corpus of sentences parsed manually to analyze new sentence. The output of this parser is an natural language parsed tree, which is important in NL to PL translation systems such as AnyCode). In this tree, each nodes will belong to one type of the Penn Treebank tags). In this section, we study the purpose of each Treebank tags and its context in NL description for PL. In total, there are 82 tags defined in). In these tags. 36 tags are frequently used in NL descriptor which belongs to 3 types: clause, phrase and word level. This section will introduce

2.2.1 Clause level tags

Clause level tags are elements for representing a grammar structure of sentences. The most frequent type of clause is the simple declarative clause which has abbreviation S. S is the basic blocks for conversation and writing). The grammar of declarative sentence usually contains subject, verb and object. An example of S is shown in).

Along with declarative sentence, NL grammar allows other types of sentences such as interrogate, imperative or exclamatory. There are 4 other types of clause defined by PennTree Bank: SBAR, SBARQ, SINV and SQ. SBAR stands for subordinate clause. It usually begin with subordinating

conjunction along with a simple declarative clause. For example, we can add "after" to "she ate breakfast" to make an example of SBAR. In NL description for PL, we can see SBAR frequently in the form of "if" clause. In example ?, "if" solved as subordinating conjunction before description of boolean condition "a is greater than zero". SBARQ stands for direct question introduced by wh-word or wh-phrase. Since NL description is usually a description about behavior of implementation, SBARQ rarely see in NL to PL translation. SINV is the inverted declarative sentence, which usually has past tense verb in its content like case in example. SQ is a question that have yes/no answer. Differ from SBARQ question, SQ sentences don't start with wh-element. An example of SBARQ is shown in case of).

2.2.2 Phrase level tags

Phrase level are usually at non-terminal level of NL tree. Each type of phrase level tags consist element at leaf node as word level. In table), we show the list of phrase level tags along with examples.

2.2.3 Word level tags

Clause level tags are node at terminal level in NL tree. The list of each word level tags and example are shown in table). In NL to PL translation, noun, verb preposition and conjunction are the most important tags used for NL description. Noun group usually represents for class name and variable name,, with plural or non-plural. Verb are used as statement trigger or method invocation description.Preposition is usually used as the connector between variables in the context of arguments in a method invocation, while conjunction responds to the connection between phrase in NL. Group of adjective and adverb are usually used for representing properties of a class or an object instance.

CHAPTER 3. METHODS AND PROCEDURES

In the previous chapter, we discuss main problems of NL to PL translation and elements of NL Parser, which are different types of tags for building an NL parse tree representation for NL sentence. These knowledge help us for gaining background to design our syntax-based approach for NL to PL translation. while NLP researches relies on analyzing NL parsed tree, we also considered NL parsed tree for NL description as the core element for translation. To handle the syntax of NL parsed tree, we relied on how programming language handled an Abstract Syntax Tree (AST) which is a representation of a source code file by building ASTVisitor. Based on ideas of ASTVisitor in PL, we designed the NLPLVisitor for NL to PL translation.

3.1 Designing a NLPL Visitor

3.1.1 Visitor Pattern

In programming language design, collections are most important data types. A collection may have instances from multiple types. Since we usually use loop for handling information from collection, it is very common that we need to write an operation or method to handle each element of collection without knowing the type of each elements. There is one way to write an operation, which used "instanceof" condition to check the type of each elements in a collection, however this way is not reflecting object oriented design.

A pattern that allows handle element of different types inside a collection in object oriented way calls Visitor pattern. In a visitor pattern, developers have the ability for representing operations on each types of elements of a collection. Besides, it allows you to write new operation without interfering the previous operations you define for types of element. In other words, visitor pattern overcomes the conflict between object structure s and algorithm which the the object structure operates on, means you can update the algorithm handling for each elements without changing the

structure of element. Along with collections, visitor pattern is also available for other data type structures such as tree structure.

One of the most useful data structure for handling source code is Abstract-Syntax- Tree (AST). In AST tree, each node in tree belong to a specific node type depend on the source code. An example of AST is shown on figure). In this example, "println" is a method invocation node which contains elements as its child node like a String literal argument. Obviously, to handle content of AST, we need to have an visitor pattern for taking operation on each type of nodes. ASTVisitor, which is developed by eclipse JDT ())is an important implementation of visitor pattern for operating the AST tree.

A design diagram of ASTVisitor is shown as follows. ASTVisitor is the name of the interface in package of eclipse JDT. It has several operation defined for each type of AST node, the most important operation is the visit() method declaration. There are set of method visit() in ASTVisitor, which each method take a specific type of ASTNode as type of its argument and each method will execute some algorithms based on type of ASTNode. To handle the operation on ASTNode on our own, we need to define a class that implements ASTVisitor and define content inside body of each visit operation. Our own visitor can be passed as an argument of ASTParser, which is used to produce the ASTNode for the whole source code and storing information of our visitor. A full class diagram of an ASTVisitor is shown on figure).

3.1.2 NLPLVisitor

Inspiring from advantages of ASTVisitor in programming language, we use a visitor pattern for design a module for dealing with the complexity of Natural Language parsed tree in NL to PL translation. NL parsed tree, to be remind, is a data structure for representing NL description in the form of a tree in which each node belongs to a specific tags in a tag sets of Penn Tree bank we discussed in the previous chapter. Similar to handling AST tree in PL, we design and implement an abstract class NLPL Visitor which we defined the basic visit method declaration for all types of NL tags. Then, we defined implementation of each visit operation relates to each type by writing a

class that implements the NLPL visitor. Since our work is for translating between two languages, we called this implemented class Translation Visitor.

Considering the translation can be done by a syntax based approach, we implement the content of each visit function as follows. First, we need to understand the intuition of each NL tags in the translation to PL. Next, we provide rules for each NL tags for translating to get elements in PL. In this step, we considered the children of each NL node and its terminal (textual value) as the most important information to decide which rules will be used for the translation of the node. The intuition of each NL tags is provided based on our observations on the NL description corpus as documentation of Java Development Kit (JDK), and the summarization of them are shown in figure). The implemented rules for each NL tags will be discussed in the next section.

3.2 Solving Problems between Natural Language and Programming Language

3.2.1 Context: Extracting Information from Variable and Literal

Context of NL description is information that can be mentioned in it. We consider surrounding code of NL description in a source code file and al Java APIs that can be accessible by a Java project that contains the description are two resources for representing the context of source code. In NLPL, given input of java project, NLPL has a module for extracting all accessible APIs in this project. They are APIs from imported library and APIs defined in each source files. The algorithm for extracting all accessible APIs is shown in figure).

In NL side,an NL description usually mentions about information of variable and literal value along with class name. We consider ambiguous types of class name is one type of indirect reference and we handle that by a reasoning module in the next section. For variables, we get information of all accessible variables in a set of variable object extract by algorithm) and check in NL description if it mentioned about a variable or not. For literal, we check if NL description contains a literal value by regular expression. We define regular expressions for string value and numeric value to do the checking.

One problems of NL parser is that it relied on probabilistic parsing, so it can be error-prone. By

looking at examples of NL descriptions and see the results of NL parser manually, we identify that keeping variable and literal in original form caused parsing with not useful information. For example, in the code showed in figure), we see that a string value "text.txt" broke into 3 parts in NL parser, while we should consider it as a single element for processing. Similarly, with example 2 on figure), the variable "a" is parsed to be a node of verb, which is not correct. To overcome this problem which cause by non-useful tags and incorrect tags, we represented variables and literal as an unique alias by the last step of preprocessing step in algorithm). An example of unique alias is shown in the example of algorithm.

3.2.2 NL Tree Transformation

The appearance of variable and literal inside NL description leads to our observation is that, in NL descriptions, some part can be considered as textual node while other parts are actually expressions like variable and literal. The expression parts can be used directly when composing to the final results for translation. This fact motivates us to transform the original parsed tree to a transformed tree with 2 types of nodes: Textual node and Expression node. This transformation allows us to embedding information specifically depends on node. An example of NL Tree transformation is shown on figure). In this example, we are able to embed information about variable a such as its type, its scope as local variable inside an Expression node. The class diagram of Textual Node and Expression is shown on figure). Both Expression node and Textual node will have information about NL tags, they keep all informations from original NL parsed tree. Since the final output for NLPL is source code which can be expressions or statement, the transformed tree served as a representation of a bridge language which connect information between source language and target language. The idea of bridge language has been applied in Natural Language translations such as for English, French or Russian).

3.2.3 Phrase: Connecting information between each Phrase in translation

We handle complexity in phrase level by the idea that we need to find a way to combine translation results of each phrases in NL description. We consider conjunction tags CC, such as "and" or "or" are two main elements that separate between each phrase in translation. In NL description, we consider that both "and" and "or" are usually used between boolean literal for expressing logical operator. In addition, "and" is usually used for combining translated results of each single phrase. We handle the "and" and the "or" conjunctions by algorithm shown in figure).

An example of different usage of "and" conjunction is shown by 3 examples in figure). In the first example, we have "and" appears between two boolean expressions that are used for comparison. In this case, due to the translated result is boolean for each single phrase, then the final translated result of "and" will be "&&" operator. In the second example, we see that the translated result for the first single clause is a constructor and the type of the constructor is the type of receiver of method API in the second clause. In this case, the combineExpression() function will mix translated results of these two phrases into a single expression. The thrid example shows a different condition compared to example 2. In this case, the constructor in the first clause doesn't have the same type with neither receiver and arguments. In this case the combineExpression() function will return 2 expressions consecutively as the final translated result.

In the first 3 sections, we propose algorithms for take advantages of context in NL to PL translation and reasoning on on complex phrases, which are two per six main problems of naturalistic programming ()). In next sections, we will introduce our solution for solving four remaining problems which relate to Indirect reference: deixis, expressiveness, anaphoric relation and ambiguity.

3.2.4 Rules for translating Natural Language elements

In the NLPLVisitor, we provide rules for handling each NL tags. The main idea is that we identify types of tags, what children tags it produce then provide rules and its terminal value. An overview of rules and function relate to each tags is shown in table).

3.2.5 Solving IR on Noun Phrase as class name

3.2.5.1 Identifying class name as Noun Phrase

3.2.5.2 Resolving types for class name

3.2.6 Solving IR on Verb Phrase as Statement Trigger

3.2.7 Solving IR on Verb Phrase as Method Invocation

3.2.8 Solving IR on Preposition as Method API

CHAPTER 4. EXPERIMENTAL

In this section, I want to answer on the research question: How well NLPL performs for translating from Natural Language comments to code.

4.1 Data Preparation

4.2 Experiment Result

No	Natural Language Comment	NLPL Top-1 result	Top-K Acc
		Expected result	
1	copy file fname to destination	FileUtils .copyFile(new File(fname) ,new File(destination))	1
		FileUtils .copyFile(new File(fname), new File(destination))	
2	load class "MyClass.class"	Thread .currentThread() .getContextClassLoader() .loadClass("MyClass .class")	1
		Thread .currentThread() .getContextClassLoader() .loadClass(MyClass .class)	
3	make file text.txt	new LineNumberReader(new InputStreamReader(new File("text .txt"))) .ready()	4
		new File(text .txt) .createNewFile()	

4	write ”hello” to file ”text.txt”	FileUtils .writeStringToFile(new File(”text .txt”) , ”hello”)	1
		FileUtils .writeStringToFile(new File(text .txt), hello)	
5	new buffered reader text.txt	new BufferedReader(new InputStream-Reader(”text .txt”))	1
		new BufferedReader(new InputStream-Reader(”text .txt”))	
6	open connection http://www.oracle.com/	new URL(”http://www .oracle .com/”) .openConnection()	1
		new URL(http://www .oracle .com/) .openConnection()	
7	create socket http://www.oracle.com/ 80	new Socket(”http://www .oracle .com/” , 80)	1
		new Socket(http://www .oracle .com/, 80)	
8	put a pair Mike , +41-345-89-23 into a map	new HashMap() .put(”Mike” , ”+41-345-89-23”)	1
		new HashMap() .put(Mike, +41-345-89-23)	
9	set thread max priority	Thread .currentThread() .setPrior-ity(Thread .MAX_PRIORITY)	1
		Thread .currentThread() .setPrior-ity(Thread .MAX_PRIORITY);	
10	set property gate.home to value http://gate.ac.uk/	new Properties() .setProperty(”gate .home” , ”http://gate .ac .uk/”)	1

		new Properties() .setProperty(gate .home, http://gate .ac .uk/)	
11	does the file 'text.txt' exist	new File("text .txt") .exists()	1
		new File("text .txt") .exists( )	
12	get thread id	Thread .currentThread() .getId()	1
		Thread .getCurrentThread() .getId()	
13	join thread	Thread .currentThread() .join()	1
		Thread .getCurrentThread() .join()	
14	delete file text.txt	FileDeleteStrategy .NORMAL .delete(new File("text .txt"))	1
		new File(text .txt) .delete()	
15	print exception ex stack trace	ex .printStackTrace()	1
		ex .printStackTrace()	
16	is text.txt directory	new File("text .txt") .isDirectory()	1
		new File(text .txt) .isDirectory()	
17	get thread stack trace	Thread .currentThread() .getStack-Trace()	1
		Thread .currentThread() .getStack-Trace()	
18	read line by line file text.txt	new LineNumberReader(new Input-StreamReader(new File("text .txt"))) .readLine()	1
		FileUtils .readLines(new File(text .txt))	
19	set time zone to GMT	Calendar .getInstance() .setTime-Zone(TimeZone .getTimeZone("GMT"))	1

		Calendar .getInstance() .setTime-Zone(TimeZone .getTimeZone(GMT))	
20	free memory	Runtime .getRuntime() .freeMemory()	1
		Runtime .getRuntime() .freeMemory()	
21	total memory	Runtime .getRuntime() .totalMemory()	1
		Runtime .getRuntime() .totalMemory()	
22	new data input stream text.txt	new DataInputStream(new FileInput-Stream("text .txt"))	1
		new DataInputStream(new FileInput-Stream(text .txt))	
23	rename file text1.txt to text2.txt	new File("text1 .txt") .renameTo(new File("text2 .txt"))	1
		new File(text1 .txt) .renameTo(new File(text2 .txt))	
24	move file text1.txt to text2.txt	FileUtils .moveFile(new File("text1 .txt") ,new File("text2 .txt"))	1
		FileUtils .moveFile(new File(text1 .txt), new File(text2 .txt))	
25	read utf from the file text.txt	FileUtils .readFileToString(new File("text .txt"))	2
		new DataInputStream(new FileInput-Stream(text .txt)) .readUTF()	
26	set thread min priority	Thread .currentThread() .setPrior-ity(Thread .MIN_PRIORITY)	1
		Thread .currentThread() .setPrior-ity(Thread .MIN PRIORITY)	

27	create panel and set layout to border	new Panel() .setLayout(new BorderLayout())	1
		new Panel() .setLayout(new BorderLayout())	
28	sort array	Arrays .sort(array)	1
		Arrays .sort(array)	
29	add label Names: to panel	new Panel() .add(new Label("Names:"))	1
		new Panel() .add(new Label(Names:))	
30	write 2015 to data output stream text.txt	new DataOutputStream(new FileOutputStream("text .txt")) .writeInt(2015)	1
		new DataOutputStream(new FileOutputStream(text .txt)) .write(2015)	
31	get date when file text.txt was last time modified	new File("text .txt") .lastModified()	>10
		new Date(new File(text .txt) .lastModified()) .getTime()	
32	check file text.txt read permission	AccessController .checkPermission(new FilePermission("text .txt" , "read"))	1
		AccessController .checkPermission(new FilePermission(text .txt, read))	
33	read lines with numbers from file text.txt	new LineNumberReader(new InputStreamReader(new File("text .txt"))) .readLine()	1

		new LineNumberReader(new InputStreamReader(new FileInputStream(text .txt))) .readLine()	
34	read from console	new BufferedReader(new InputStreamReader(System .in)) .read()	1
		new BufferedReader(new InputStreamReader(System .in)) .readLine()	
35	is file text.txt data available	new DataInputStream(new FileInputStream(new File("text .txt"))) .available()	1
		new DataInputStream(new FileInputStream(text .txt)) .available()	
36	get double value x	Integer .valueOf(x) .doubleValue()	>10
		Double .valueOf(x) .doubleValue()	
37	write object o to file output stream data.obj"	new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream("data .obj"))) .writeObject(o)	1
		new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(data .obj))) .writeObject(o)	
38	create bit set and set its 5th element to true	new BitSet() .set(5 , true)	1
		new BitSet() .set(5,true)	
39	accept request on port 80	new ServerSocket(80) .accept()	1
		new ServerSocket(80) .accept()	

40	get thread group	Thread .currentThread() .getThread-Group()	1
		Thread .currentThread() .getThread-Group()	
41	create panel and set layout to grid	new Panel() .setLayout(new GridLay-out())	>10
		new Panel() .setLayout(new Grid-BagLayout())	
42	get screen size	Toolkit .getDefaultToolkit() .getScreen-Size()	1
		Toolkit .getDefaultToolkit() .getScreen-Size()	
43	get splash screen graphics	? .getSplashScreen()	>10
		SplashScreen .getSplashScreen() .cre-ateGraphics()	
44	get display refresh rate	GraphicsEnvironment .getLocal-GraphicsEnvironment() .getDefault-ScreenDevice() .getDisplayMode() .getRefreshRate()	1
		GraphicsEnvironment .getLocal-GraphicsEnvironment() .getDefault-ScreenDevice() .getDisplayMode() .getRefreshRate()	
45	get keystroke modifiers	keystroke .getModifiers()	1
		KeyEvent .getKeyModifiers-Text(keystroke .getModifiers())	

46	generate RSA private key	new Thread(”RSA”) .yield()	>10
		KeyPairGenerator .getInstance(RSA) .generateKeyPair() .getPrivate()	
47	reverse list	Collections .reverseOrder()	>10
		Collections .reverse(list)	
48	intersection of rectangle 4 5 with rectangle 3 2	? .intersection(?)	>10
		new Rectangle(5, 4) .intersection(new Rectangle(3, 2))	
49	set cursor over label to hand	new AffineTransform() .setToIdentity()	>10
		label .setCursor(Cursor .getPredefined-Cursor(Cursor .HAND CURSOR))	
50	read big integer from console	new BufferedReader(new InputStream-Reader(System .in)) .read()	3
		new Scanner(System .in) .nextBigInte-ger()	
51	delete file text.txt when JVM ter-minates	new File(”text .txt”) .delete()	>10
		new File(text .txt) .deleteOnExit()	
52	get date instance for Germany	DateFormat .getDateTimeInstance() .getDateInstance()	>10
		DateFormat .getDate-TimeInstance(DateFormat .MEDIUM,DateFormat .MEDIUM, Locale .GERMANY)	

4.3 Result Analysis

4.4 NLPL Online IDE

K	AnyCode	NLPL	Top-K of AnyCode	Top-K of NLPL
1	32	39	61.54%	75.00%
2	39	40	75.00%	76.92%
3	41	41	78.85%	78.85%
4	42	42	80.77%	80.77%
5	45	42	86.54%	80.77%

CHAPTER 5. SUMMARY AND DISCUSSION

5.1 Related Works on Syntax-based approach in translation between Natural Languages

5.2 Related Works on Inferring Implementation

REFERENCES

Barone, A. V. M. and Sennrich, R. (2017). A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *CoRR*, abs/1707.02275.

Gvero, T. and Kuncak, V. (2015). Synthesizing java expressions from free-form queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 416–432, New York, NY, USA. ACM.

Knuth, D. (2009). Xstring file. http://www.literateprogramming.com/string.w.

Knuth, D. E. (1984). Literate programming. *Comput. J.*, 27(2):97–111.

Luong, M., Brevdo, E., and Zhao, R. (2017). Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*.

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2013). Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA. ACM.

Phan, H. (2018). Nlpl web tool. http://209.124.64.139:8080/NLPLWebTool/.

Phan, H., Nguyen, H. A., Nguyen, T. N., and Rajan, H. (2017). Statistical learning for inference between implementations and documentation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ICSE-NIER '17, pages 27–30, Piscataway, NJ, USA. IEEE Press.

Pulido-Prieto, O. and Juárez-Martínez, U. (2017). A survey of naturalistic programming technologies. *ACM Comput. Surv.*, 50(5):70:1–70:35.

Socher, R., Bauer, J., Manning, C. D., and Andrew Y., N. (2013). Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 455–465. Association for Computational Linguistics.

StackExchange (2010). Why isn't literate programming mainstream? https://softwareengineering.stackexchange.com/questions/811/why-isnt-literate-programming-mainstream.

Young, T., Hazarika, D., Poria, S., and Cambria, E. (2018). Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75.

APPENDIX A. ADDITIONAL MATERIAL

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

More stuff

Supplemental material.

APPENDIX B. STATISTICAL RESULTS

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Supplemental Statistics

More stuff.