Creating C# Programming Corpus using ANTLR4 for Non-Native English-Speaking Students

Satoshi Numata
Department of Digital Games, Osaka Electro-Communication University.
1130-70 Shijonawate Kiyotaki, Osaka, Japan.
E-mail: numata@osakac.ac.jp
Tel: +81-72-876-5103

ABSTRACT

In programming, naming classes, functions, and variables is important for making the code readable and easy to maintain. As most of the programming languages are designed based on English, non-native English-speaking programming learners have to learn English and the programming language simultaneously, which can be a barrier to learners, especially in Asian countries, to start programming because grammar and words in Asian languages are quite different. Native English speakers can easily search, read, and learn from many sample codes published on the Internet. I propose a system for automatically creating a corpus from a massive number of sample codes found on the Internet using ANTLR4, which is a powerful parser generator using grammar definitions. I expect such a corpus can help non-native English-speaking learners and many professional developers .

Keywords: C#, Programming Education, English Learning, ANTLR4

1. Introduction

In programming, naming is an important process for creating readable and maintainable program codes. In most cases, programming can be separated into designing algorithms and naming variables and functions to implement the algorithms. Therefore, naming can consist of half the programming process. Once we select a good name for a variable, we will be able to verify whether the variable is updated appropriately for its role in the code.

According to Boswell and Foucher (2011), we can separate "surface-level improvements" into three parts: selecting good names, writing good comments, and formatting codes neatly. Regarding good comments, we sometimes can remove comments if variable and function names are more descriptive. Where codes are often modified, comments can easily be left unchanged and it starts to mislead the content of the code. Therefore, the naming process is the most important at the "surface-level" if we put the code-formatting process together with algorithm design.

1.1. Programming Language and English

As mentioned above, descriptive names of variables and functions can help in the understanding of code behavior. For example, if the statement in the code below can be read directly as an English sentence, i.e., if the target string starts with "http://", it will start to download the page according to the URL of the target string.

```
string targetString = "http://example.com/test.txt";
if (targetString.startsWith("http://")) {
   DownloadPage(targetString);
}
```

Because most popular programming languages were developed in the United States, most of the grammatical features stand on English expressions, especially in the C-family of programming languages such as C, C++, C#, and Java.

Therefore, knowledge of English is one of the key elements and required for better programming understanding. This can be a barrier to non-native English-speaking programming learners. Native English speakers can simply read sample codes and understand the basic ideas of the programming

language. To learn English for daily life, we can find books, such as "Everyday English Words", that explain basic words suitable for daily use. However, there are few learning materials for the English used in programming languages. When we search source codes on the Internet, we can find that the programming elements, such as classes, functions, and variables, are sometimes named using local languages. However, this makes it difficult to understand the code structure because local languages often have different grammatical structures for the programming statements. It is also sometimes difficult to figure out whether a variable expresses one entity or several because some languages do not differentiate singular and plural forms whereas most English words clearly differentiate them.

We can find services on the Internet that help us name programming entities using automatic translation systems, and some students in my department are using such services. Because the class, structure, function, and variable names are often very short (mostly two or three words), the name of the source language does not provide enough information as context for translation in most cases, so it is difficult to obtain the correct translated names. It is important to teach English together with programming to enable students to write readable and understandable codes.

Thus, it is important to learn basic English words used in source codes as well as the programming ideas simultaneously for non-native English speakers. This is why I suggest the importance of a coding corpus for programming languages. I propose a system, for creating a corpus for C# programming language using ANTLR4 (2012).

1.2. Parsing Source Codes using ANTLR4

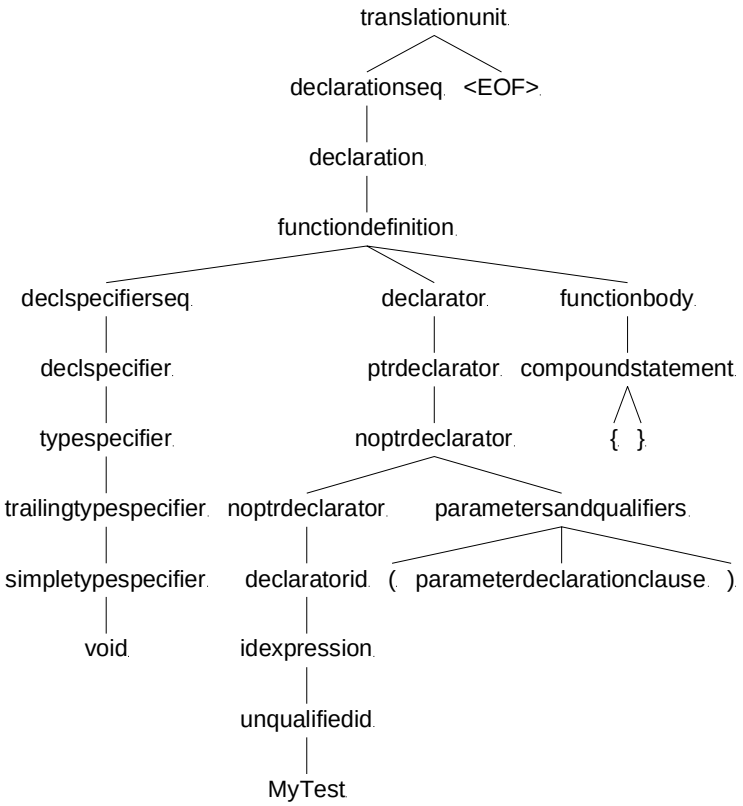
ANTLR4 is a parser generator that takes LL(*) grammars and outputs source codes of a lexical analyzer and parser. The ANTLR project also provides a collection of grammars of many popular programming languages at the antlr/grammars-v4 on GitHub (2012), including C, C++, Java, C#, Ruby, Python, and Swift. The coding corpus for many programming languages can be created using those grammars with ANTLR4.

2. Grammars and Words of Programming Languages

We have to consider two limitations for choosing the target programming language of a corpus: the simplicity of the grammar and the number of sample projects found on GitHub repositories. From the grammatical viewpoint, some programming languages are not suitable for analyzing to create coding corpora.

2.1. Grammatical Structure of C and C++

We first examine C and C++ languages. The C language has evolved by adding many features ad hoc, and it is difficult to simply trace the structure of the code. According to Stroustrup (1994), C++ was designed to keep its compatibility with C; thus, C++ inherits the difficulty of C, and is also difficult to trace the code structure. Figure 1 shows how the C++ code "void MyFunc() {}" is parsed by the parser generated by ANTLR4 using the C++14 grammar. You can see the deeply nested



structure from the function definition node.

Figure 1: *Example of C++ code analysis. Function definition node has deep descendant nodes.*

2.2. Grammatical Structure of C#

Regarding C#, it is possible to simply trace the structure of the code. Figure 2 shows an example how the C# code "class Test { void MyTest() {} }" is parsed. You can see the method (function) declaration node has shallow and simple descendant nodes, and it is very easy to extract the function name from its structure. Hence, we can conclude that the newer C-family of languages such as C# and Java, are better for creating a coding corpus.

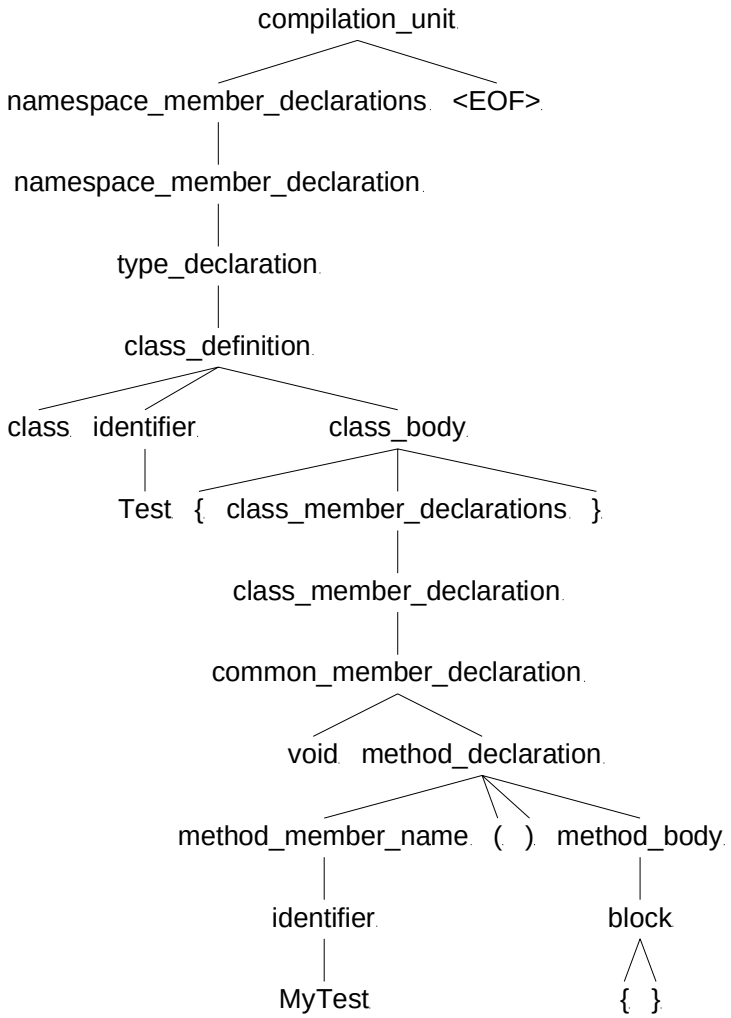


Figure 2: *Example of C# code analysis. Method declaration node has only descendant nodes of depth of two before leaf keywords*.

Many open source projects are using GitHub as a hub for publication and a source code management system. GitHub enables us to search these projects by words, programming languages, starred numbers, and update frequency. For example, we can search popular game-related projects written in C# by specifying *"game language:C# stars:>=1000"* as a search word. There were 11 large game-related C# projects starred over 1000 times to examine the source codes that include a total of 14,127 files and 2,165,577 lines of codes. I chose C# games because they are suitable for beginning learners and I wanted to examine programming words in a specific area first.

2.3. Naming Analysis using ANTLR4

For C# grammar, ANTLR4 outputs a lexical analyzer, parser, and listener of the parser. The listener is notified each time the parser finds a new element from a source code. Because of the well-structured and clear grammar of C#, using directives, namespaces, classes, structures, fields (variables), methods (functions), method arguments, local variables in methods can be found. When the listener is notified of the appearance of those coding entities, that information is put on a stack. When an identifier appears, that information will be popped out from the stack and combined with the identifier. Such identifiers can now be logged as named-coding entities.

3. Corpus-Creating System

3.1. System Overview

The proposed system is composed of the three subsystems shown in Figure 3 for creating a coding corpus. The listener of a parser outputs coding entities into an XML file. The XML file keeps the code structure (File > Namespace > Class > Field, Method > Local Variable) because these entities are output in order of the parsing precedence. Therefore, word information should be flattened before the examination. The word-flattening subsystem is thus prepared for flattening the words kept in the grammatical structure into a Comma-Separated Values (CSV) file. After the flattening phase, the examination subsystem can load the flatten data and carry out statistical analysis.

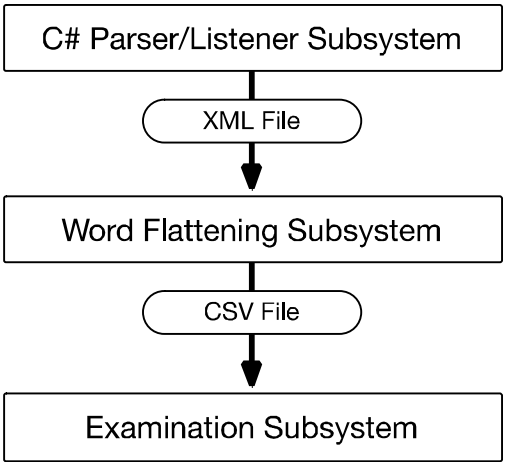


Figure 3: *Overview of three subsystems of proposed system*

For example, let us assume the following C# code:

```
using System.*;
public class PlayerAnimation {
   void StartWithDelay(float delay) {
      int startCount = 0;
      bool hasFinished = false;
      ...
   }
}
```

This code is lexically analyzed and parsed using the listener. The listener finds coding entities and outputs them as an XML file as follows (a "compilation_unit" node corresponds to a source code file):

```
<compilation_unit>
  <using target="System.*" />
  <class name="PlayerAnimation">
    <method-decl name="StartWithDelay">
      <arg name="delay" type="float" />
      <block>
        <local-var name="startCount" type="int" />
        <local-var name="hasFinished" type="bool" />
      ...
  </class>
</compilation_unit>
```

The word data are flattened as follows (leading numbers and colons indicate the line numbers):

```
1: class,PlayerAnimation,[File]sample/PlayerAnimation.cs
2: method,StartWithDelay,[Class]PlayerAnimation,[File]sample/
    PlayerAnimation.cs
3: local-var,startCount,type=int,[Block],[Method]StartWithDelay,
   [Class]PlayerAnimation,[File]sample/PlayerAnimation.cs
4: local-var,hasFinished,type=bool,[Block],[Method]RunActivity,
   [Class]PlayerAnimation,[File]sample/PlayerAnimation.cs
...
```

The first element in each line is an entity-type specifier, and the second element is the name. If the entity is a class field or local variable of a method, the third element is the type information such as 'int', 'float', and 'string'. All ancestor entities are listed afterward for making it possible to trace the origins of the named entities.

3.2. Normalization of Underscore and Capitalization Style

Though there is little difference between how the entity names are selected in most cases, the coding conventions differ in each project. The difference mostly appears on the variable or function names, and some include underscores and have capitalization. For instance, an integer value that indicates effect type can be expressed as a variable with the name "effectType", "effect_type", or "EffectType." The name can be "m_effectType" in some cases for clearly declaring that is a field member of a class. Therefore, normalization should be carried out for each extracted word. I present an algorithm for normalizing a name (Algorithm 1).

Algorithm 1: *Normalization of name with underscore and capitalization style*

$R \leftarrow \emptyset$; $p \leftarrow name.len - 1$
while $p \geq 0$ **do**
    $c \leftarrow name(p)$
    $p \leftarrow p - 1$
    while $c =$ '_' $\wedge$ $p \geq 0$ **do**
        $c \leftarrow name(p)$
        $p \leftarrow p - 1$
    end while
    if $p < 0$ **then** $R \leftarrow R \cup \{ lower(c) \}$; **exit**
    $s \leftarrow c$
    while $p \geq 0$ **do**

```
            d ← name(p)
            p ← p - 1
            if c.type = d.type then
                s ← d + s
            else
                case c.type of
                    Lower:
                        case d.type of
                            Upper: R ← R ∪ { lower(d + s) }; s ← ∅; exit
                            Otherwise: R ← R ∪ { lower(s) }; s ← ∅; exit
                        end case
                    Upper: R ← R ∪ { lower(s) } ; p ← p + 1; s ← ∅; exit
                    Number: p ← p + 1; s ← ∅; exit
                    Otherwise: R ← R ∪ { lower(s) }; s ← ∅; exit
                end case
            end if
        end while
        if s.len > 0 then
            R ← R ∪ { lower(s) }
        end if
    end while
```

In this algorithm description, *R* indicates the normalized set that holds every component separated by underscores or capitalizations, and *name* is a string that contains the entity name. It scans the name from the end to the beginning, so that a name, such as "OpenGLManager", can be correctly separated into "open", "gl", and "manager." If we scan this from the beginning, capital characters in the middle will be connected and "glm" will be extracted. Few programmers might name the same manager as "OpenGLmanager", from which "lmanager" will be extracted with the proposed algorithm, but this should be rare, so I assumed that it can be ignored in statistical analysis. Numbers and underscores in the name are removed with this algorithm.

3.3. System Implementation

I implemented the proposed system as the application shown in Figure 4. This application is implemented using Java. By checking the types of coding entities, it can search or rank words used in actual projects in GitHub.

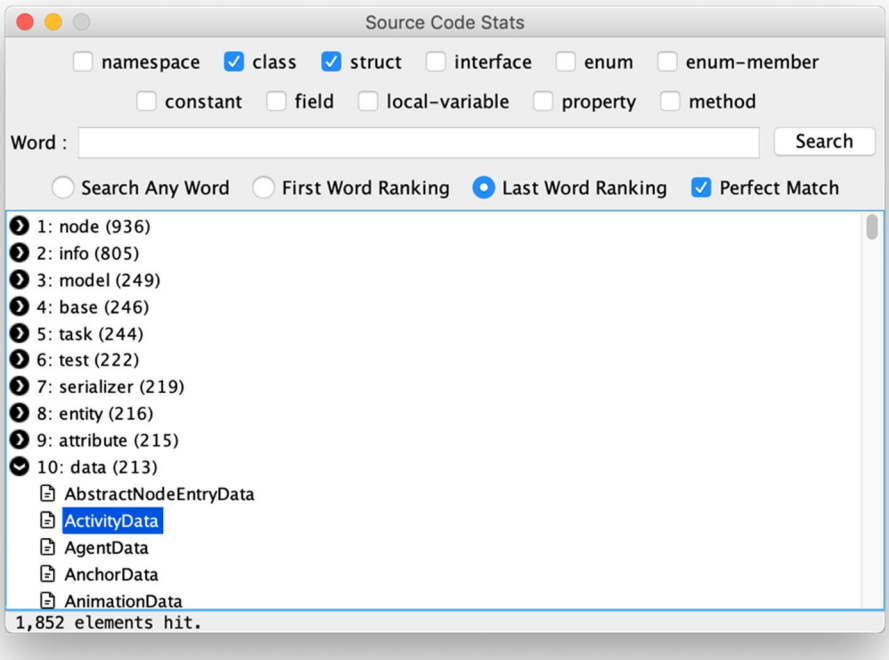


Figure 4: *Experimental implementation of system. Ranking of last word used in class and structured names are listed as tree.*

As described in Section 2.2, it loads the word data that were statistically analyzed from 14,127 files and 2,165,577 lines of codes over 11 repositories of game-related projects written in C#. It loads 336,799 names of coding entities. Table 1 shows the number of each coding entity.

Table 1: *Number of extracted names for coding entities*

Class	Structure	Interface	Enums	Enum-Member	Method	Field	Local Variable	Constant
18,800	1,220	1,257	1,750	13,553	96,682	50,280	99,495	6,884

For example, the ranking of the last words used in class and structure names is (number in parentheses shows how many times the word appears in source codes) 1-Node (936), 2-Info (805),

3-Model (249), 4-Base (246), 5-Task (244), 6-Test (222), 7-Serializer (219), 8-Entity (216), 9-Attribute (215), and 10-Data (213). The ranking of the first words used in methods is 1-get (8343), 2-on (3670), 3-create (2972), 4-update (2868), 5-set (2761), 6-load (2170), 7-equals (1771), 8-to (1726), 9-test (1709), and10-add (1641).

Some words listed above indicate the feature of the programming language I examined, such as "Serializer", "on", "equals", and "to". Both rankings have the word "test" because the Test-Driven Development has been more widely used recently.

4. Conclusion

I proposed a system for creating a coding corpus for C# programming language using parsers generated from ANTLR4. I expect such a corpus can help non-native English speakers learn English effectively for understanding programming styles during their lessons.

I examined game-related projects in a specific area written in C# for this study. I will continue to work with other project categories other than games and other languages to investigate the tendency of such categories and programming languages.

References

Boswell, D., & Foucher, T. (2011). The Art of Readable Code: Simple and Practical Techniques for Writing Better Code. O'Reilly Media, USA.

ANTLR4 (2012). Retrieved from http://www.antlr.org

antlr/grammars-v4 on GitHub (2012). Retrieved from https://github.com/antlr/grammars-v4

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley Professional, USA.