Java2Pseudo: Java to Pseudo Code Translator a Pilot Study

Heetae Cho[1], Seonah Lee[2]

[1] *Department of AI Convergence Engineering, Gyeongsang National University, 501 Jinju-daero, Jinju-si, Gyeongsangnam-do, Republic of Korea*

[2] *Department of Aerospace and Software Engineering, Gyeongsang National University, 501 Jinju-daero, Jinju-si, Gyeongsangnam-do, Republic of Korea*

Abstract

Novice programmers may not quickly understand a new or unfamiliar programming language. In order to help them understand the source code, existing studies have proposed approaches that translate a programming language into a pseudo-code. However, to the best of our knowledge, no studies have proposed translating Java, one of the well-known object-oriented programming languages, to pseudo-code. Many novice programmers learn the Java language because it is relatively less complicated than C++ and versatile in terms of practical use. Furthermore, the educational curriculum includes a Java course frequently. In this paper, we propose an approach to translate Java into pseudo-code at the code fragment level. We expect the proposed approach could help novice programmers to learn the Java language with the translated pseudo-code.

Keywords

Java, Program Comprehension, Pseudo-code

1. Introduction

A pseudo-code presents the logic of an algorithm in a natural language imitating a programming language. The pseudo-code forms of natural language are more readable and understandable than interpreting the programming language directly. Therefore, novice programmers could easily understand the meaning of the actual source code through pseudo-code.

For this reason, several studies have proposed approaches to translating source code to pseudo-code [1, 2, 3, 4, 5, 6, 7, 8]. However, no studies were translating Java source code to pseudo-code. Furthermore, most studies used machine learning techniques [1, 2, 3, 4, 6, 8], which require much effort to gather pseudo-code data corresponding to source code.

Although machine learning techniques are strong forward, we hypothesize the programming language is formal language enough to generate pseudo-code using rules like a compiler process.

In this paper, we propose a translation approach to line-by-line translating Java source code to pseudo-code using context for matching code patterns and a template for translating matched patterns to pseudo-code. This approach does not need the effort to gather data like machine learning techniques.

This paper is organized as follows. Section 2 describes related works for our study. Section 3 introduces our approach. Section 4 demonstrates the demo results. Section 5 describes the discussion. Finally, section 6 concludes this paper.

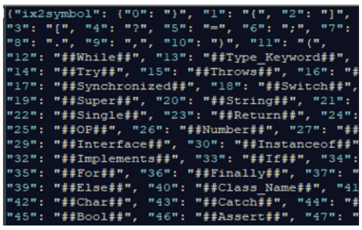
2. Related Works

While existing studies translate source code to pseudo-code, most studies [1, 2, 3, 4, 6, 8] targeted Python source code. These studies also used machine-learning techniques.

Oda et al.[1] used statistical machine translation techniques that parse-based machine translation and tree-to-string machine translation to generate pseudo-code from python source code. Xu et al.[2] and Alhefdhi et al.[6] used sequence-to-sequence and attention approaches. They encode python source code with LSTM encoder and decode it to pseudo-code with LSTM decoder. Yang et al.[3]

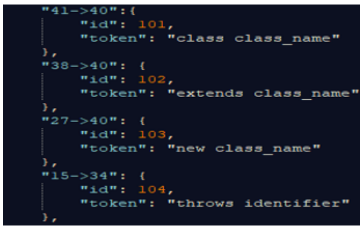
1st International Workshop on Intelligent Software Engineering, December 06, 2022, Busan, Republic of Korea

cht3205@gnu.ac.kr (H. Cho); saleese@gnu.ac.kr (S. Lee)



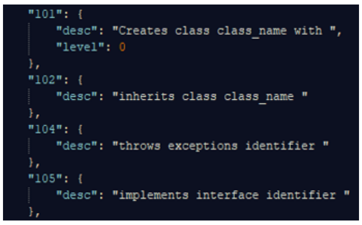
```
{"ix2symbol": {"0": "}", "1": "{", "2": "]",
"3": "[", "4": "?", "5": "=", "6": ";", "7":
"8": ".", "9": ",", "10": ")", "11": "(",
"12": "##While##", "13": "##Type_Keyword##",
"14": "##Try##", "15": "##Throws##", "16": "#
"17": "##Synchronized##", "18": "##Switch##",
"19": "##Super##", "20": "##String##", "21":
"22": "##Single##", "23": "##Return##", "24":
"25": "##OP##", "26": "##Number##", "27": "##
"29": "##Interface##", "30": "##Instanceof##"
"32": "##Implements##", "33": "##If##", "34":
"35": "##For##", "36": "##Finally##", "37": "
"39": "##Else##", "40": "##Class_Name##", "41
"42": "##Char##", "43": "##Catch##", "44": "#
"45": "##Bool##", "46": "##Assert##", "47": "
```

Figure 1: Pre-defined Symbols



```
"41->40":{
    "id": 101,
    "token": "class class_name"
},
"38->40": {
    "id": 102,
    "token": "extends class_name"
},
"27->40": {
    "id": 103,
    "token": "new class_name"
},
"15->34": {
    "id": 104,
    "token": "throws identifier"
},
```

Figure 2: Pre-defined Patters



```
"101": {
    "desc": "Creates class class_name with ",
    "level": 0
},
"102": {
    "desc": "inherits class class_name "
},
"104": {
    "desc": "throws exceptions identifier "
},
"105": {
    "desc": "implements interface identifier "
},
```

Figure 3: Pre-defined Templates

used CNN and transformer architecture[9]. They first extract code features using the CNN model. Then the extracted features are fed to the transformer architecture to generate pseudo-code. Gad et al.[4], and Alokla et al.[8] also used transformer architecture. They first tokenize source code with their own rules, and vectorize them and apply position embedding. Then the embedded source codes are fed to the transformer architecture to generate pseudo-code.

These studies treated source code as natural language. However, their approaches have a potential shortcoming in that such an approach regards distinct contexts despite the same ones with different identifiers. For instance, from a pseudo-code perspective, the contexts "int a" and "int b" are equally treated as variable declarations. Treated as natural language, however, different vectors are created because of the different words 'a' and 'b'.

Two study used contexts and templates similar to our study [5, 7]. However, they target JavaScript and Python which has different contexts of a code line to Java.

3. Approach

Our approach, inspired by a compiler, to translating Java source code to pseudo-code is as follows:

- step-1) Remove all comments and blank lines and tokenizes the source code
- step-2) Changes code tokens to pre-defined symbols and keeps the original code tokens of each symbol
- step-3) Changes the symbols to specific symbols by checking the original tokens
- step-4) Finds patterns line-by-line from pre-defined patterns through the longest symbol sequence-first match
- step-5) Generates pseudo-code of the matched patterns through pre-defined pseudo-code template

Where the samples of pre-defined symbols, patterns, and templates are shown in Figures 1 - 3 respectively. First, the pre-defined symbols, as shown in Figure 1, represent the roles of the code tokens and assign an ID for each token. Second, the patterns shown in Figure 2 express a unique sequence of symbols and IDs. Finally, the templates, as shown in Figure 3, describe each pattern.

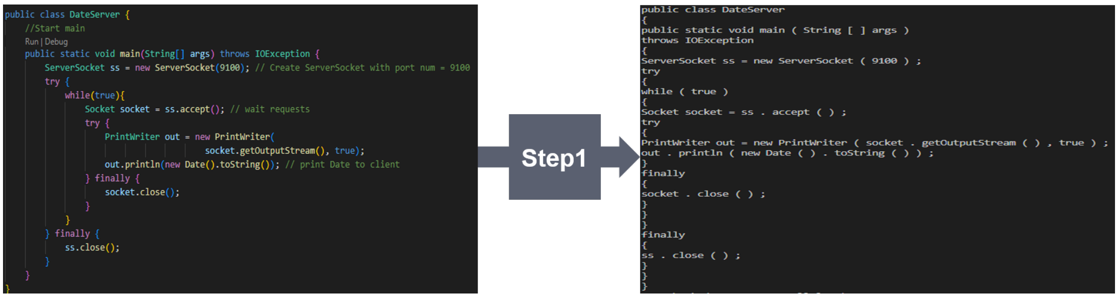
4. Demonstration

4.1. Approach Demonstration

To demonstrate our approach, we first depict following the steps. For step 1, as shown in Figure 4, we remove comments and blank lines from the entered source code and tokenize them. For step 2, shown in Figure 5, we replace all the code tokens with pre-defined symbols. For step 3, we replace the symbols with unique IDs, as shown in Figure 6. For step 4, as shown in Figure 7, we use the pre-defined patterns to find the longest pattern from left to right in the symbol ID sequence, and if found, iteratively search from the following ID. If the pattern is not found, leave the symbol ID and search again from the following ID. Finally, as shown in Figure 8, we generate the pseudo-code with the pre-defined templates.

4.2. Prototype

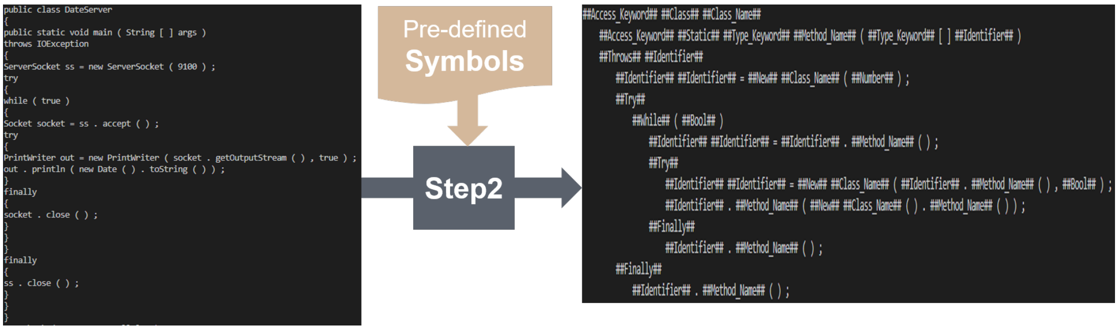
We implemented a prototype of our translation approach as a web. Figures 9-11 show the demonstration of our approach. Figure 9 shows the textarea



```
public class DateServer {
    //Start main
    Run | Debug
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(9100); // Create ServerSocket with port num = 9100
        try {
            while(true){
                Socket socket = ss.accept(); // wait requests
                try {
                    PrintWriter out = new PrintWriter(
                                        socket.getOutputStream(), true);
                    out.println(new Date().toString()); // print Date to client
                } finally {
                    socket.close();
                }
            }
        } finally {
            ss.close();
        }
    }
}
```

```
public class DateServer
{
public static void main ( String [ ] args )
throws IOException
{
ServerSocket ss = new ServerSocket ( 9100 ) ;
try
{
while ( true )
{
Socket socket = ss . accept ( ) ;
try
{
PrintWriter out = new PrintWriter ( socket . getOutputStream ( ) , true ) ;
out . println ( new Date ( ) . toString ( ) ) ;
}
finally
{
socket . close ( ) ;
}
}
}
finally
{
ss . close ( ) ;
}
}
}
```

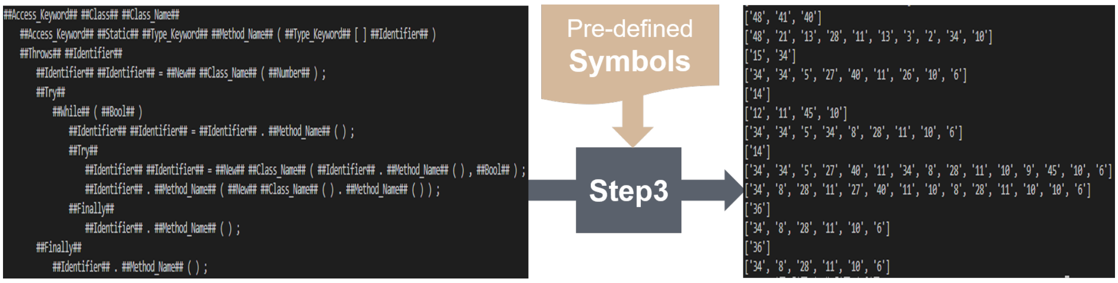
Figure 4: Step-1) Tokenizes the source code



```
public class DateServer
{
public static void main ( String [ ] args )
throws IOException
{
ServerSocket ss = new ServerSocket ( 9100 ) ;
try
{
while ( true )
{
Socket socket = ss . accept ( ) ;
try
{
PrintWriter out = new PrintWriter ( socket . getOutputStream ( ) , true ) ;
out . println ( new Date ( ) . toString ( ) ) ;
}
finally
{
socket . close ( ) ;
}
}
}
finally
{
ss . close ( ) ;
}
}
}
```

```
##Access_Keyword## ##Class## ##Class_Name##
  ##Access_Keyword## ##Static## ##Type_Keyword## ##Method_Name## ( ##Type_Keyword## [ ] ##Identifier## )
  ##Throws## ##Identifier##
    ##Identifier## ##Identifier## = ##New## ##Class_Name## ( ##Number## ) ;
    ##Try##
      ##While## ( ##Bool## )
        ##Identifier## ##Identifier## = ##Identifier## . ##Method_Name## ( ) ;
        ##Try##
          ##Identifier## ##Identifier## = ##New## ##Class_Name## ( ##Identifier## . ##Method_Name## ( ) , ##Bool## ) ;
          ##Identifier## . ##Method_Name## ( ##New## ##Class_Name## ( ) . ##Method_Name## ( ) ) ;
        ##Finally##
          ##Identifier## . ##Method_Name## ( ) ;
    ##Finally##
      ##Identifier## . ##Method_Name## ( ) ;
```

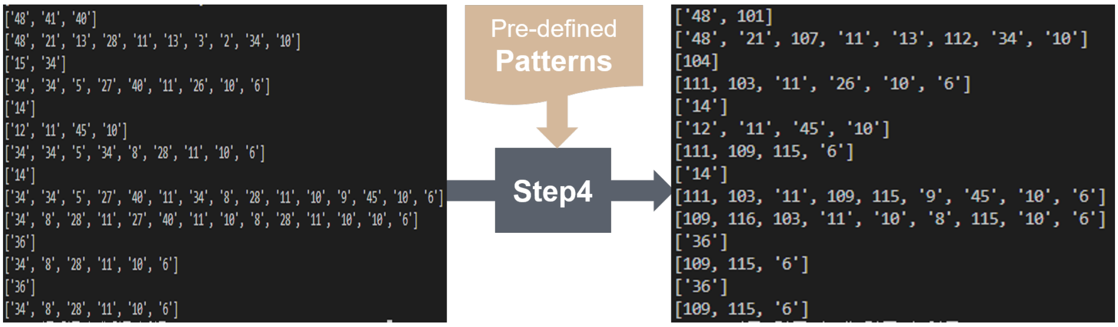
Figure 5: Step-2) Changes code tokens to pre-defined symbols



```
##Access_Keyword## ##Class## ##Class_Name##
  ##Access_Keyword## ##Static## ##Type_Keyword## ##Method_Name## ( ##Type_Keyword## [ ] ##Identifier## )
  ##Throws## ##Identifier##
    ##Identifier## ##Identifier## = ##New## ##Class_Name## ( ##Number## ) ;
    ##Try##
      ##While## ( ##Bool## )
        ##Identifier## ##Identifier## = ##Identifier## . ##Method_Name## ( ) ;
        ##Try##
          ##Identifier## ##Identifier## = ##New## ##Class_Name## ( ##Identifier## . ##Method_Name## ( ) , ##Bool## ) ;
          ##Identifier## . ##Method_Name## ( ##New## ##Class_Name## ( ) . ##Method_Name## ( ) ) ;
        ##Finally##
          ##Identifier## . ##Method_Name## ( ) ;
    ##Finally##
      ##Identifier## . ##Method_Name## ( ) ;
```

```
['48', '41', '40']
['48', '21', '13', '28', '11', '13', '3', '2', '34', '10']
['15', '34']
['34', '34', '5', '27', '40', '11', '26', '10', '6']
['14']
['12', '11', '45', '10']
['34', '34', '5', '34', '8', '28', '11', '10', '6']
['14']
['34', '34', '5', '27', '40', '11', '34', '8', '28', '11', '10', '9', '45', '10', '6']
['34', '8', '28', '11', '27', '40', '11', '10', '8', '28', '11', '10', '10', '6']
['36']
['34', '8', '28', '11', '10', '6']
['36']
['34', '8', '28', '11', '10', '6']
```

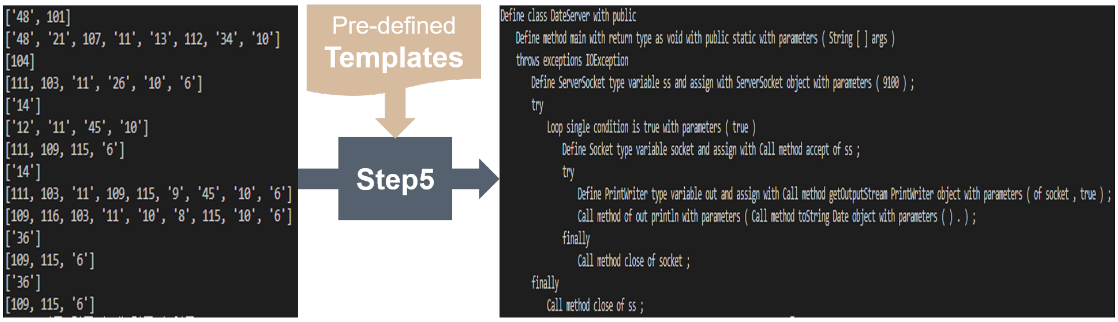
Figure 6: Step-3) Changes the symbols to unique IDs



```
['48', '41', '40']
['48', '21', '13', '28', '11', '13', '3', '2', '34', '10']
['15', '34']
['34', '34', '5', '27', '40', '11', '26', '10', '6']
['14']
['12', '11', '45', '10']
['34', '34', '5', '34', '8', '28', '11', '10', '6']
['14']
['34', '34', '5', '27', '40', '11', '34', '8', '28', '11', '10', '9', '45', '10', '6']
['34', '8', '28', '11', '27', '40', '11', '10', '8', '28', '11', '10', '10', '6']
['36']
['34', '8', '28', '11', '10', '6']
['36']
['34', '8', '28', '11', '10', '6']
```

```
['48', 101]
['48', '21', 107, '11', '13', 112, '34', '10']
[104]
[111, 103, '11', '26', '10', '6']
['14']
['12', '11', '45', '10']
[111, 109, 115, '6']
['14']
[111, 103, '11', 109, 115, '9', '45', '10', '6']
[109, 116, 103, '11', '10', '8', 115, '10', '6']
['36']
[109, 115, '6']
['36']
[109, 115, '6']
```

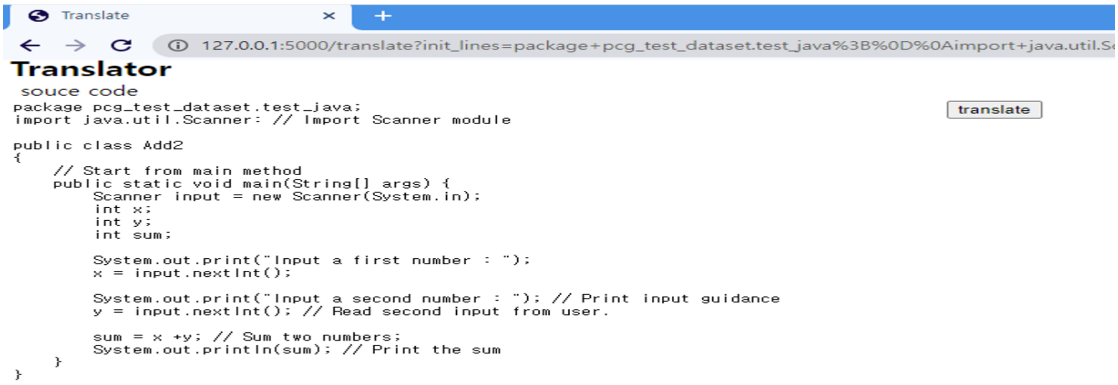
Figure 7: Step-4) Finds patterns with the pre-defined patterns



```
['48', 101]
['48', '21', 107, '11', '13', 112, '34', '10']
[104]
[111, 103, '11', '26', '10', '6']
['14']
['12', '11', '45', '10']
[111, 109, 115, '6']
['14']
[111, 103, '11', 109, 115, '9', '45', '10', '6']
[109, 116, 103, '11', '10', '8', 115, '10', '6']
['36']
[109, 115, '6']
['36']
[109, 115, '6']
```

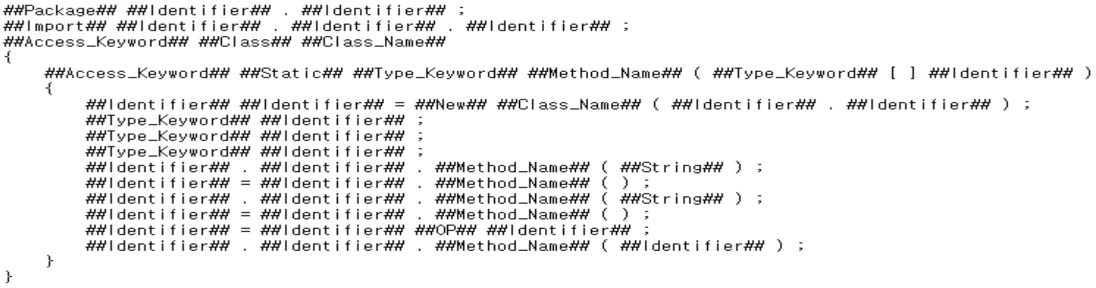
```
Define class DateServer with public
  Define method main with return type as void with public static with parameters ( String [ ] args )
  throws exceptions IOException
    Define ServerSocket type variable ss and assign with ServerSocket object with parameters ( 9100 ) ;
    try
      Loop single condition is true with parameters ( true )
        Define Socket type variable socket and assign with Call method accept of ss ;
        try
          Define PrintWriter type variable out and assign with Call method getOutputStream PrintWriter object with parameters ( of socket , true ) ;
          Call method of out println with parameters ( Call method toString Date object with parameters ( ) . ) ;
        finally
          Call method close of socket ;
    finally
      Call method close of ss ;
```

Figure 8: Step-5) Generates the pseudo-code with the pre-defined templates


Translate
127.0.0.1:5000/translate?init_lines=package+pcg_test_dataset.test_java%3B%0D%0Aimport+java.util.S
Translator
souce code
translate

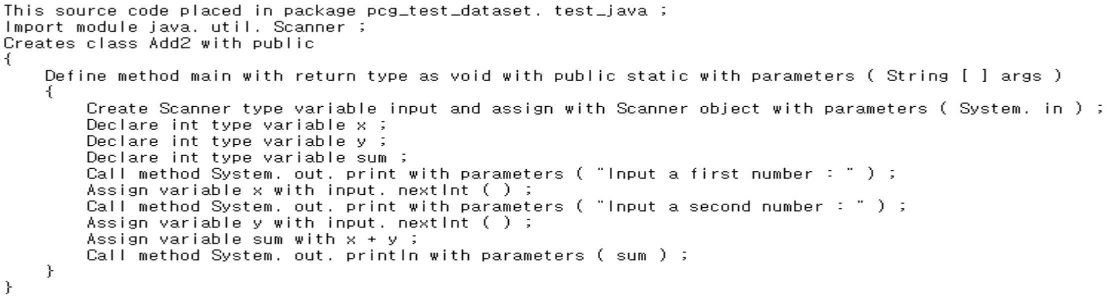
```
package pcg_test_dataset.test_java;
import java.util.Scanner: // Import Scanner module

public class Add2
{
    // Start from main method
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int x;
        int y;
        int sum;

        System.out.print("Input a first number : ");
        x = input.nextInt();

        System.out.print("Input a second number : "); // Print input guidance
        y = input.nextInt(); // Read second input from user.

        sum = x +y; // Sum two numbers;
        System.out.println(sum); // Print the sum
    }
}
```

Figure 9: Step-1) Tokenizes the source code



```
##Package## ##Identifier## . ##Identifier## ;
##Import## ##Identifier## . ##Identifier## . ##Identifier## ;
##Access_Keyword## ##Class## ##Class_Name##
{
    ##Access_Keyword## ##Static## ##Type_Keyword## ##Method_Name## ( ##Type_Keyword## [ ] ##Identifier## )
    {
        ##Identifier## ##Identifier## = ##New## ##Class_Name## ( ##Identifier## . ##Identifier## ) ;
        ##Type_Keyword## ##Identifier## ;
        ##Type_Keyword## ##Identifier## ;
        ##Type_Keyword## ##Identifier## ;
        ##Identifier## . ##Identifier## . ##Method_Name## ( ##String## ) ;
        ##Identifier## = ##Identifier## . ##Method_Name## ( ) ;
        ##Identifier## . ##Identifier## . ##Method_Name## ( ##String## ) ;
        ##Identifier## = ##Identifier## . ##Method_Name## ( ) ;
        ##Identifier## = ##Identifier## ##OP## ##Identifier## ;
        ##Identifier## . ##Identifier## . ##Method_Name## ( ##Identifier## ) ;
    }
}
```

Figure 10: Step-2) Changes code tokens to pre-defined symbols



```
This source code placed in package pcg_test_dataset. test_java ;
Import module java. util. Scanner ;
Creates class Add2 with public
{
    Define method main with return type as void with public static with parameters ( String [ ] args )
    {
        Create Scanner type variable input and assign with Scanner object with parameters ( System. in ) ;
        Declare int type variable x ;
        Declare int type variable y ;
        Declare int type variable sum ;
        Call method System. out. print with parameters ( "Input a first number : " ) ;
        Assign variable x with input. nextInt ( ) ;
        Call method System. out. print with parameters ( "Input a second number : " ) ;
        Assign variable y with input. nextInt ( ) ;
        Assign variable sum with x + y ;
        Call method System. out. println with parameters ( sum ) ;
    }
}
```

Figure 11: Step-3) Changes the symbols to unique IDs

for inputting the Java source code. When clicking the translate button, the inputted source code changes to symbolic code through the approach's steps 1 to 4. For example, source code line "Scanner scan = new Scanner (System.in);" is changed to "##Identifier## ##Identifier## = ##Keyword## ##Identifier## (##Identifier## . ##Identifier ##) ;", as shown in Figure 10. Then, the symbols are transformed into specific symbols. After that, By using the longest symbol sequence-first match, our approach replaces the symbolic code with symbol sequence patterns based on pre-defined symbol patterns. For example, the symbolic code above changes to ['(id id =)', '(keyword id)', '((id . id)' '( ))'] which as pre-defined symbol sequence. Finally, pseudo-code generates by extracting pseudo-code for each pattern from a pre-defined pseudo-code template, as shown in figure 11. For example, the (id id =) changed to "create Scanner type object variable input and assign with", the (keyword id) changed to "Scanner object", and (id . id) changed to "with parameters (System . in)".

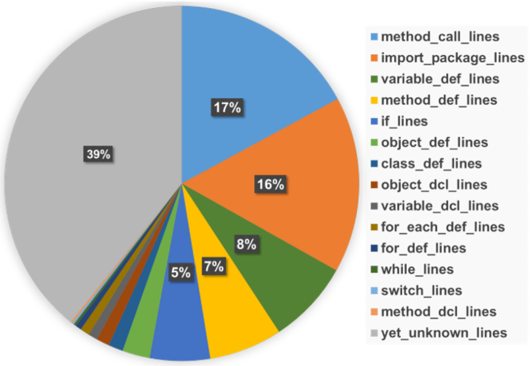


Figure 12: Ratio of the source code lines

5. Discussion

To check the feasibility of our approach, we analyzed the Java source code to create the symbol sequence patterns and the pseudo-code template. We gathered 10 Apache projects (i.e., Flink, Beam, Hbase, Cassandra, Tomcat, Storm, Rocketmq, Jmeter, Zookeeper, Zeppelin) from Github. The projects contained total of 37,679 Java code files, written 7,485,340 lines, including comments, blanks, etc.

We got around 3,251,000 SLOC for those lines by removing comments and blank lines and merging multi-lines into single lines. After that, we got around 210,000 symbolic code lines by tokenizing the source code lines and changing tokens to symbols. Then, we checked the possibility of using the regular expression to distinguish the role of the symbolic code line. As a result, as shown in Figure 12, we identified around 60% of the roles of source code lines (around 1.6 million lines) with only 14 regular expressions for symbolic code lines. The top 5 kinds of code line roles are as follows:

- method call lines (17.1%) (e.g., method();)
- package/import lines (16.0%) (e.g., import java.util.*;)
- variable definition lines (7.6%) (e.g., int a=1; except object)
- method definition lines (6.6%) (e.g., int method ())
- if or else-if lines (5.5%)

This result shows that the specific symbol sequences can identify the roles of each line.

6. Conclusion

In this paper, we analyzed contexts of the java source code line and implemented a translation approach from Java source code to pseudo-code with symbolic code.

This prototype has several threats and limitations. First, the source code tokenizer implementation could be incorrect because the language we used for our implementation, Python, cannot distinguish escape sequence \” as separated characters ‘ \’ and ‘ ” ’ while reading files. Second, complicated or missing statements cannot translate appropriately since not all id sequence patterns have been checked. Finally, many keywords appear as they are because the scope of the pseudo-code is not clearly defined.

To overcome the threats and limitations, we will develop using another programming language and adopt the automata theory for checking the patterns.

References

[1] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, S. Nakamura, Learning to generate pseudo-code from source code using statistical machine translation, in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 574–584.

[2] S. Xu, Y. Xiong, Automatic generation of pseudocode with attention seq2seq model, in: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2018, pp. 711–712.

[3] G. Yang, Y. Zhou, X. Chen, C. Yu, Fine-grained pseudo-code generation method via code feature extraction and transformer, in: 2021 28th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2021, pp. 213–222.

[4] W. Gad, A. Alokla, W. Nazih, M. Aref, A. Salem, Dlbt: deep learning-based transformer to generate pseudo-code from source code, Cmc-Comput. Mater. Contin 70 (2022) 3117–3132.

[5] A. Barmpoutis, Learning programming languages as shortcuts to natural language token replacements, in: Proceedings of the 18th Koli Calling International Conference on Computing Education Research, 2018, pp. 1–10.

[6] A. Alhefdhi, H. K. Dam, H. Hata, A. Ghose, Generating pseudo-code from source code using deep learning, in: 2018 25th Australasian Software Engineering Conference (ASWEC), IEEE, 2018, pp. 21–25.

[7] S. Rai, R. C. Belwal, A. Gupta, Is the corpus ready for machine translation? a case study with python to pseudo-code corpus, Arabian Journal for Science and Engineering (2022) 1–14.

[8] A. Alokla, W. Gad, W. Nazih, M. Aref, A.-B. Salem, Retrieval-based transformer pseudocode generation, Mathematics 10 (2022) 604.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).