**Question**

We have two following code clones. Perform the following tasks with the best of your knowledge with compilers and Java programming language. You can reuse any compiler tool (e.g., Java tokenizer and parser) if you can.

Q1. Represent each code fragment as a sequence of lexical tokens in Java. Are those two sequences identical?

Q2. Using those token sequences to represent each code fragment as a vector (each dimension is a token type).

Q3. Represent each code fragment as an abstract syntax tree. Are those two trees identical or highly similar?

Q4. Using those two trees to represent each code fragment by a vector (each dimension is a node type)

Q5. Keep only textual content in each code fragment and tokenize them. For example:

if (input.endsWith("ed"))

is converted into

if input ends with ed

Are two code fragments still highly similar?

// (\*v\*) ED ->  
if (input.endsWith("ed")) {  
 String stem = input.substring(0, input.length() - 2);  
 String letterTypes = getLetterTypes(stem);  
 if (letterTypes.contains("v")) {  
 return step1b2(stem);  
 }  
 return input;  
}

// (\*v\*) ING ->  
if (input.endsWith("ing")) {  
 String stem = input.substring(0, input.length() - 3);  
 String letterTypes = getLetterTypes(stem);  
 if (letterTypes.contains("v")) {  
 return step1b2(stem);  
 }  
 return input;  
}

**Note:**

**To run the Java code for above question I have used notepad++ terminal. We need to have the Java environment in local machine prior to download notepad ++.**

**The link to download notepad++ is** [**https://notepad-plus-plus.org/downloads/**](https://notepad-plus-plus.org/downloads/) **. This link is for Windows OS, as my local machine is Windows OS supported.**

**Steps to run the Java program using command Prompt:**

* **Open the code in notepad ++ terminal.**
* **Copy the file path of the respective code.**
* **Open the command prompt and pass through the copied path.**
* **Now run the command as javac Name of Java Class.java**
* **Now run the command as java Name of Java Class.**

Solutions:

Q1. Represent each code fragment as a sequence of lexical tokens in Java. Are those two sequences identical?

Below is the Java code to represent each code fragment as a sequence of lexical tokens in Java.

import java.util.ArrayList;

import java.util.List;

public class tokenizer {

public static void main(String[] args) {

String code1 = "if (input.endsWith(\"ed\")) { String stem = input.substring(0, input.length() - 2); String letterTypes = getLetterTypes(stem); if (letterTypes.contains(\"v\")) { return step1b2(stem); } return input; }";

String code2 = "if (input.endsWith(\"ing\")) { String stem = input.substring(0, input.length() - 3); String letterTypes = getLetterTypes(stem); if (letterTypes.contains(\"v\")) { return step1b2(stem); } return input; }";

List<String> tokens1 = tokenize(code1);

List<String> tokens2 = tokenize(code2);

System.out.println("Code Fragment 1 Tokens: " + tokens1);

System.out.println("Code Fragment 2 Tokens: " + tokens2);

boolean areIdentical = tokens1.equals(tokens2);

System.out.println("Are the token sequences identical? " + areIdentical);

}

private static List<String> tokenize(String input) {

List<String> tokens = new ArrayList<>();

String[] splitTokens = input.split("\\s+");

for (String token : splitTokens) {

if (!token.isEmpty()) {

tokens.add(token);

}

}

return tokens;

}

private static String getLetterTypes(String stem) {

return "";

}

private static String step1b2(String stem) {

return "";

}

}

Below snapshot is the execution of the above code:

A screenshot of a computer

Description automatically generated

From the above output, we can state that the token sequences of given code fragments are not identical when they are represented in lexical tokens format.

When we investigate the above code first, we import the required classes from the java.util package such as ArrayList and List to create and manipulate the lists in Java. Later the code defines a Tokenization class with a main method and the two Java code fragments (code1 and code2) are provided as strings. Now, the above code involves, tokenizing each code fragment by breaking it into individual words or symbols and using a Scanner to scan and collect tokens from the code and print the resulting tokens for both code fragments. In the tokenize method the provided each code fragment was operated and split into an array of tokens using the ‘split’ method with regular expression. Lastly, the code compares the two lists (tokens1 and tokens2) using the equals method to see if the token sequences for codes 1 and 2 are the same.

Q2. Using those token sequences to represent each code fragment as a vector (each dimension is a token type).

The Java code to represent each code fragment as a vector is as follows:

import java.util.\*;

public class VectorRepresentation {

public static void main(String[] args) {

String code1 = "if (input.endsWith(\"ed\")) { String stem = input.substring(0, input.length() - 2); String letterTypes = getLetterTypes(stem); if (letterTypes.contains(\"v\")) return step1b2(stem); return input; }";

String code2 = "if (input.endsWith(\"ing\")) { String stem = input.substring(0, input.length() - 3); String letterTypes = getLetterTypes(stem); if (letterTypes.contains(\"v\")) return step1b2(stem); return input; }";

List<String> tokens1 = tokenize(code1);

List<String> tokens2 = tokenize(code2);

System.out.println("Code 1 Tokens: " + tokens1);

System.out.println("Code 2 Tokens: " + tokens2);

Map<String, Integer> vector1 = createVector(tokens1);

Map<String, Integer> vector2 = createVector(tokens2);

System.out.println("Code 1 Vector: " + vector1);

System.out.println("Code 2 Vector: " + vector2);

}

private static List<String> tokenize(String input) {

List<String> tokens = new ArrayList<>(Arrays.asList(input.split("\\s+|(?<=[^\\w\\s])|(?=[^\\w\\s])")));

tokens.removeIf(String::isEmpty);

return tokens;

}

private static Map<String, Integer> createVector(List<String> tokens) {

Map<String, Integer> vector = new HashMap<>();

for (String token : tokens) vector.put(token, vector.getOrDefault(token, 0) + 1);

return vector;

}

private static String getLetterTypes(String stem) {

return "";

}

private static String step1b2(String stem) {

return "";

}

}

Output:

A black and white screen

Description automatically generated

When we look into the above ode at first, we include java.util package to perform operations such as tokenization and vector representation. Later, we derive a Java class named VectorRepresentation within a main method to tokenize and represent two specified code segments as vectors depending on the frequency of specific lexical tokens. Later, in the main method, the two Java code fragments (code1 and code2) are provided as strings, and the two code fragments are tokenized into the list of strings using the tokenize method. Later, the createVector method creates a vector representation by counting token occurrences where the ‘HashMap’ data structure is used to store the frequency of each unique token in the provided list of tokens. The resulting vector representations are printed for both code fragments.

Q3. Represent each code fragment as an abstract syntax tree. Are those two trees identical or highly similar?

Code Snippet1

// (\*v\*) ED ->  
if (input.endsWith("ed")) {  
 String stem = input.substring(0, input.length() - 2);  
 String letterTypes = getLetterTypes(stem);  
 if (letterTypes.contains("v")) {  
 return step1b2(stem);  
 }  
 return input;  
}

// (\*v\*) ING ->  
if (input.endsWith("ing")) {  
 String stem = input.substring(0, input.length() - 3);  
 String letterTypes = getLetterTypes(stem);  
 if (letterTypes.contains("v")) {  
 return step1b2(stem);  
 }  
 return input;  
}

Code Snippet2

**Note:**

Below diagrams are built in the draw.io website for the ease of use of symbols.

This is the Abstract Syntax Tree for the code snippet 1

A diagram of a flowchart

Description automatically generated

Below is the Abstract Syntax Tree for the code snippet 2:

A diagram of a flowchart

Description automatically generated

The two abstract syntax trees (ASTs) for Code 1 and Code 2 are highly similar but not identical. The overall structure and the types of nodes are the same in both trees, reflecting the similarity in the code structure. However, there are differences in the conditions and specific values used in the code.

Here are the main points of similarity and difference:

**Similarities:**

* Both trees have the same overall structure with an **if** statement, conditions, and similar code blocks.
* The structures inside the code blocks are identical, involving **substring()**, **contains("v")**, **return**, **getLetterTypes()**, and **step1b2()**.

**Differences:**

* The conditions of the **if** statements are different: one checks for the string ending with "ed," and the other checks for the string ending with "ing."
* The values used in the **endsWith()** conditions are different.

In conclusion, the trees are highly similar due to the shared code structure and logic, but they are not identical because the conditions and specific values differ.

Q4. Using those two trees to represent each code fragment by a vector (each dimension is a node type)

The Java code to represent each code fragment by a vector using the two above trees

import java.util.HashMap;

import java.util.Map;

public class ASTVectorRepresentation {

public static void main(String[] args) {

String astCode1 = "(if (endsWith input \"ed\") (block (assign stem (substring input 0 (- (length input) 2))) (assign letterTypes (getLetterTypes stem)) (if (contains letterTypes \"v\") (return (step1b2 stem)) (return input))))";

String astCode2 = "(if (endsWith input \"ing\") (block (assign stem1 (substring input 0 (- (length input) 3))) (assign letterTypes1 (getLetterTypes stem1)) (if (contains letterTypes1 \"v\") (return (step1b2 stem1)) (return input))))";

Map<String, Integer> vectorCode1 = createVector(astCode1);

Map<String, Integer> vectorCode2 = createVector(astCode2);

System.out.println("Code 1 Vector: " + vectorCode1);

System.out.println("Code 2 Vector: " + vectorCode2);

}

private static Map<String, Integer> createVector(String ast) {

Map<String, Integer> vector = new HashMap<>();

String[] nodes = ast.split(" ");

for (String node : nodes) {

vector.put(node, vector.getOrDefault(node, 0) + 1);

}

return vector;

}

}

The output:

A screenshot of a computer

Description automatically generated

In the above code, the abstract syntax trees (ASTs) of Code 1 and Code 2 are essentially trees that represent the structure of the code. Here, we've expressed them as simple strings where different operations or actions in the code are separated by spaces. We have a method called createVector that takes an AST string (like the ones we created for Code 1 and Code 2) and turns it into a vector. This vector is like a list where each item is a type of operation or action from the code. Inside the createVector method, we're essentially counting how many times each operation or action appears in the code. This count is stored in a data structure called a HashMap.

In the main method, we use the createVector method to get vectors for both Code 1 and Code 2.

We then print out these vectors so that we can visually compare them.

Q5. Keep only textual content in each code fragment and tokenize them. For example:

if (input.endsWith("ed"))

is converted into

if input ends with ed

The Java code to tokenize only textual content of given two code fragments is a s follows:

import java.util.\*;

public class TextTransformation {

public static void main(String[] args) {

String code1 = " v ED " +

"if input.endsWith ed " +

" String stem input.substring 0, input.length - 2 "+

" String letterTypes getLetterTypes stem "+

" if letterTypes.contains v " +

" return step1b2 stem "+

" return input ";

String code2 = " v ING " +

"if input.endsWith ing " +

" String stem input.substring 0, input.length - 3" +

" String letterTypes getLetterTypes stem " +

" if (letterTypes.contains v " +

" return step1b2 stem" +

" return input" ;

String transformedTextCode1 = transformText(code1);

String transformedTextCode2 = transformText(code2);

List<String> tokensCode1 = tokenize(transformedTextCode1);

List<String> tokensCode2 = tokenize(transformedTextCode2);

System.out.println("Transformed Text Code 1: " + transformedTextCode1);

System.out.println("Transformed Text Code 2: " + transformedTextCode2);

System.out.println("\nTokenization");

System.out.println("Code 1 Tokens: " + tokensCode1);

System.out.println("Code 2 Tokens: " + tokensCode2);

}

private static String transformText(String code) {

String transformedText = code.replaceAll("[^a-zA-Z\\s]", "");

return transformedText;

}

private static List<String> tokenize(String input) {

List<String> tokens = new ArrayList<>();

Scanner scanner = new Scanner(input);

while (scanner.hasNext()) {

tokens.add(scanner.next());

}

return tokens;

}

private static String getLetterTypes(String stem) {

return "";

}

private static String step1b2(String stem) {

return "";

}

}

The output:

A screenshot of a computer

Description automatically generated

Here in the above code at first, we create a method named TextTransformation where within the main method the two Java code fragments (code1 and code2) are given as strings.

The transformText method takes a code fragment and removes non-alphabetic characters, keeping only letters and spaces.

The tokenize method uses a Scanner to tokenize the transformed text, creating a list of words (tokens).

The main method applies the text transformation and tokenization to both code fragments.

It prints the transformed text and tokens for each code fragment.

The primary idea is to simplify the code by focusing on the textual content, removing symbols and special characters, and then tokenize the simplified text for comparison. The approach involves regular expressions for text transformation and a Scanner for tokenization.

The transformed text for both Code 1 and Code 2 reveals a high degree of similarity. Both code fragments follow a similar structure and perform analogous operations, such as checking the ending of the input, extracting a substring, and evaluating letter types.

Therefore, based on the textual content after transformation, we can conclude that the two code fragments are still highly similar.

**Note:**

1. The name of the Java class in the submission files to execute 1st question is tokenizer.java.
2. The name of the java class in the submission files to execute 2nd question is VectorRepresentation.java .
3. The name of the Java class in the submission files to execute the 4th question is ASTVectorRepresentation.java.
4. The name of the Java class in the submission files to execute the 5th question is TextTransformation.java .