Lab 7

Subject: Software Engineering

Subject code: IT-314

Student Name: 202201466

Student Id: 202201466

Code link: https://github.com/mzazon/coreinterpreter

I. PROGRAM INSPECTION: An Error Checklist for Inspections

Category A: Data Reference Errors

1. **Does a referenced variable have a value that is unset or uninitialized?**

   Ans: Yes: Variables like symbolList[symbolIndex].ID and symbolList[symbolIndex].SYMBOL_INIT_FLAG may not be initialized before being accessed.

2. **For all array references, is each subscript value within the defined bounds of the corresponding dimension?**

   Ans: No: The subscript symbolIndex in symbolList[symbolIndex] could potentially exceed the bounds if the index is not managed properly, but it is generally handled safely in the code.

3. **For all array references, does each subscript have an integer value?**

   Ans: Yes: All array indices (like symbolIndex) are integers.

4. **For all references through pointer or reference variables, is the referenced memory currently allocated?**

   Ans: Yes: The parser ensures memory is allocated before usage, such as using new ParseTree().

5. **When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names?**

   Ans: No: This case doesn't appear to apply to this code.

6. **Does a variable's value have a type or attribute other than what the compiler expects?**

   Ans: No: Variables appear to be referenced with the expected types.

7. **Are there any explicit or implicit addressing problems?**

   Ans: No: There is no indication of explicit addressing issues in this code.

8. **If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects?**

   Ans: Yes: Pointer usage is consistent with expected attributes, though memory management (freeing memory) could be an issue.

9. **If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?**

   Ans: Yes: Structures like ParseTree and Token are consistently defined.

10. **When indexing into a string, are the limits of the string off by-one errors in indexing operations or in subscript references to arrays?**

    Ans: No: String indexing issues don't appear in this code.

11. **For object-oriented languages, are all inheritance requirements met in the implementing class?**

    Ans: No: This program doesn't implement object-oriented inheritance.

Category B: Data-Declaration Errors

1. **Have all variables been explicitly declared?**

   Ans: Yes: Variables are explicitly declared in the code.

2. **If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood?**

Ans: Yes: Attributes are clear, especially for basic types like int and ParseTree*.

3. **Where a variable is initialized in a declarative statement, is it properly initialized?**

Ans: No: Variables like symbolList[symbolIndex].ID are not initialized immediately and can cause issues later.

4. **Is each variable assigned the correct length and data type?**

Ans: Yes: Data types seem appropriate, though the lack of initialization for arrays like symbolList is an issue.

5. **Is the initialization of a variable consistent with its memory type?**

Ans: Yes: Initialization is generally consistent with memory types, but uninitialized data in some cases could cause runtime issues.

6. **Are there any variables with similar names (e.g., VOLT and VOLTS)?**

Ans: No: There are no confusingly similar variable names.

Category C: Computation Errors

1. **Are there any computations using variables having inconsistent (such as non-arithmetic) data types?**

Ans: No: The code doesn't perform complex arithmetic operations or involve type mismatches. It mainly deals with parsing tokens and recursive function calls, so there are no obvious computation errors.

2. **Are there any mixed-mode computations (e.g., addition of a floating-point variable to an integer)?**

Ans: No: There are no mixed-mode computations in the code, as it's mainly handling tokens and parsing structures.

3. **Are there any computations using variables having the same data type but different lengths?**

Ans: No: This issue is not applicable since the code does not involve arrays or data structures with variable lengths in calculations.

4. **Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?**

   Ans: No: There is no evidence of incorrect assignments related to data type mismatches.

5. **Is an overflow or underflow expression possible during the computation of an expression?**

   Ans: No: The code does not include mathematical operations that could cause overflow or underflow.

6. **Is it possible for the divisor in a division operation to be zero?**

   Ans: No: The code doesn't involve division operations.

7. **Inaccuracy due to base-2 representation?**

   Ans: No: The code doesn't involve floating-point operations, so base-2 inaccuracies aren't applicable.

8. **Can a variable go outside the meaningful range?**

   Ans: Yes: symbolIndex could exceed the size of symbolList (5000), leading to out-of-bounds errors if not properly managed.

9. **For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?**

   Ans: Yes: In the few cases where expressions are evaluated (such as comparisons), operator precedence seems correct.

10. **Are there any invalid uses of integer arithmetic, particularly divisions?**

    Ans: No: There is no arithmetic involving divisions in this code.

Category D: Comparison Errors

1. **Are there any comparisons between variables having different data types (e.g., comparing a character string to an address, date, or number)?**

Ans: No: The code compares tokens and strings, but they are of consistent types.

2. **Are there any mixed-mode comparisons or comparisons between variables of different lengths?**

   Ans: No: Mixed-mode comparisons are not present.

3. **Are the comparison operators correct?**

   Ans: Yes: The comparison operators used, like in conditional checks, are appropriately selected.

4. **Does each Boolean expression state what it is supposed to state?**

   Ans: Yes: The Boolean expressions in conditional checks are clear and correctly represent the intended logic.

5. **Are the operands of a Boolean operator Boolean?**

   Ans: Yes: The operands in Boolean expressions are logically consistent.

6. **Are there any comparisons between fractional or floating-point numbers that are represented in base-2 by the underlying machine?**

   Ans: No: The code doesn't involve floating-point arithmetic or comparisons.

7. **For expressions containing more than one Boolean operator, are the assumptions about the order of evaluation and precedence of operators correct?**

   Ans: Yes: The Boolean operators are used in simple conditions, and there is no ambiguity in precedence.

8. **Does the way in which the compiler evaluates Boolean expressions affect the program?**

   Ans: No: The code relies on short-circuit evaluation, which is handled consistently across compilers for conditions like if statements.

Category E: Control-Flow Errors

1. **If the program contains a multiway branch (e.g., a computed GO TO), can the index variable exceed the number of branch possibilities?**

Ans: No: There are no multiway branches like computed GO TOs in the code. The program relies on structured control flow (if, else if, while).

2. **Will every loop eventually terminate?**

Ans: Yes: The loops (recursions) in the program seem well-structured and should terminate, assuming valid input. However, recursion depth can be an issue if the input size grows excessively.

3. **Will the program module or subroutine eventually terminate?**

Ans: Yes: The code should terminate after parsing the input and building the parse tree.

4. **Is it possible that because of the conditions upon entry, a loop will never execute?**

Ans: Yes: Recursive calls could potentially not execute in some cases where input tokens don't match expected values.

5. **For a loop controlled by both iteration and a Boolean condition, what are the consequences of loop fall-through?**

Ans: No: This case doesn't apply, as there are no explicit loops, only recursive function calls.

6. **Are there any off-by-one errors (e.g., too many or too few iterations)?**

Ans: No: The code doesn't rely on loop indexing, so off-by-one errors are not an issue.

7. **If the language contains a concept of statement groups or code blocks, is there an explicit while for each group and do the do's correspond to their appropriate groups?**

Ans: Yes: Code blocks are properly structured, and there are no mismatches between opening and closing control structures.

8. **Are there any non-exhaustive decisions (e.g., unhandled cases in multiway branches)?**

Ans: Yes: Error handling in some places could be more exhaustive. For instance, ThrowParserError() is used generically, but more specific error handling might be helpful for debugging purposes.

Category F: Interface Errors

1. **Does the number of parameters received by this module equal the number of arguments sent, and is the order correct?**

Ans: Yes: The number of parameters matches the number of arguments passed in each function call, and the order of parameters is consistent throughout the code.

2. **Do the attributes (e.g., data type and size) of each parameter match the attributes of each corresponding argument?**

Ans: Yes: The data types and sizes of parameters and arguments are correctly matched, ensuring proper function operation without type mismatches.

3. **Does the units system of each parameter match the units system of each corresponding argument?**

Ans: No: There are no unit conversions (such as degrees to radians) in this code, so this issue does not apply.

4. **Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?**

Ans: Yes: The number of arguments passed between functions is consistent with the number of expected parameters, ensuring no argument mismatch.

5. **Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?**

Ans: Yes: The attributes (data type and size) of arguments transmitted between modules are correctly aligned with the corresponding parameters.

6. **Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?**

Ans: No: The code does not involve any unit conversions, so this is not relevant.

7. **If built-in functions are invoked, are the number, attributes, and order of the arguments correct?**

Ans: Yes: Built-in and custom functions are invoked with the correct number, attributes, and order of arguments.

8. **Does a subroutine alter a parameter that is intended to be only an input value?**

   Ans: No: Input-only parameters are not altered within the subroutines. The parameters are used appropriately without unintended modifications.

9. **If global variables are present, do they have the same definition and attributes in all modules that reference them?**

   Ans: Yes: Global variables like symbolList are consistently defined and referenced across different modules without discrepancies in their attributes.

Category G: Input / Output Errors

1. **If files are explicitly declared, are their attributes correct?**

   Ans: Yes: The file used in the Scanner class is correctly declared for reading tokens.

2. **Are the attributes on the file's OPEN statement correct?**

   Ans: Yes: The file is opened properly within the Scanner class for reading purposes.

3. **Is there sufficient memory available to hold the file your program will read?**

   Ans: Yes: Memory usage is minimal, so there should be no issues with file size handling.

4. **Have all files been opened before use?**

   Ans: Yes: The file is opened before attempting to read tokens.

5. **Have all files been closed after use?**

   Ans: No: The code does not explicitly close the file after use, which could lead to resource leaks.

6. **Are end-of-file conditions detected and handled correctly?**

   Ans: Yes: End-of-file conditions are indirectly handled through token advancement in the Advance() function.

7. **Are I/O error conditions handled correctly?**

Ans: No: There is no explicit error handling for file I/O failures, such as failed file opening or reading errors.

8. **Are there spelling or grammatical errors in any text that is printed or displayed by the program?**

   Ans: No: The printed error messages like "*** FATAL PARSER ERROR ***" are clear and free of spelling or grammatical errors.

Category H: Other Checks

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**

   Ans: Yes: All variables in the code appear to be used appropriately, with no unused or single-use variables.

2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**

   Ans: Yes: The attributes of variables, such as types for symbolList and symbolIndex, are explicitly defined, and there are no unexpected default attributes.

3. **If the program compiled successfully, but the computer produced one or more “warning” or “informational” messages, check each one carefully.**

   Ans: Yes: If warnings are generated, they should be reviewed, especially regarding memory leaks and potential uninitialized variables like symbolList[symbolIndex].ID.

4. **Is the program or module sufficiently robust? Does it check its input for validity?**

   Ans: No: The program relies on ThrowParserError() for error handling, but it could be more robust with more specific input validation and error messages.

5. **Is there a function missing from the program?**

   Ans: No: The program seems to include all necessary functions for parsing, though error handling could be improved.

Program Inspection:

1. **How many errors are there in the program? Mention the errors you have identified.**

   Ans: There are several errors in the program:

   - Uninitialized variables: Some variables like symbolList[symbolIndex].ID are used without being initialized.
   - Memory leaks: Dynamic memory (e.g., new ParseTree()) is allocated but not freed.
   - Error handling: The generic ThrowParserError() terminates the program abruptly without releasing resources or providing detailed error information.

2. **Which category of program inspection would you find more effective?**

Ans: Category A (Data Reference Errors) and Category E (Control-Flow Errors) are the most effective for this code, as they help catch uninitialized variables, memory leaks, and recursion depth issues.

3. **Which type of error are you not able to identify using the program inspection?**

Ans: Performance issues and runtime errors such as deep recursion or excessive memory use may not be easily identified through static inspection. These would be better detected using dynamic analysis tools or debugging.

4. **Is the program inspection technique worth applying?**

Ans: Yes: Program inspection is valuable for identifying logic, memory, and error handling issues early. It helps in catching common issues before runtime, making the code more robust and maintainable.

II. CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code

1) Armstrong

A. Program Inspection

1. The program contains an error related to the calculation of the remainder, which has been identified and fixed.
2. The most appropriate category of program inspection for this code is Category C: Computation Errors, since the issue involves the calculation of the remainder, a specific type of computation error.
3. Program inspection does not address debugging-related issues; it does not identify problems such as breakpoints or runtime errors, including logic errors.
4. The program inspection technique is useful for identifying and resolving problems related to code structure and computation errors.

B. Debugging

1. The program contains an error related to the calculation of the remainder, which has been identified previously.
2. To resolve this issue, set a breakpoint at the location where the remainder is computed to verify that it's calculated accurately. Step through the code to monitor the values of variables and expressions during execution.

3. The corrected executable code is as follows:

```
// Armstrong Number

class Armstrong {

public static void main(String args[]) {

int num = Integer.parseInt(args[0]);

int n = num; // used to check at the last time

int check = 0, remainder;

while (num > 0) {

remainder = num % 10;

check = check + (int) Math.pow(remainder, 3);
num = num / 10;

}

if (check == n)

System.out.println(n + " is an Armstrong Number");

else

System.out.println(n + " is not an Armstrong Number");

}

}
```

2) GCD and LCM

A. Program Inspection

1. The program contains two errors:
2. Error 1: In the gcd function, the while loop condition should be `while(a % b != 0)` instead of `while(a % b == 0)` to accurately calculate the GCD.
3. Error 2: In the lcm function, there is a logical error. The approach used to compute the LCM is flawed and could lead to an infinite loop.
4. For this code, the most relevant category of program inspection is Category C: Computation Errors, as it contains computation errors in both the gcd and lcm functions.
5. Program inspection cannot identify runtime problems or logical errors; it is unable to detect issues such as infinite loops.
6. The program inspection technique is beneficial for identifying and resolving computation-related issues.

B. Debugging

1. There are two errors in the program as mentioned above.

2. To fix these errors:

3. For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.

4. For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.

5. The corrected executable code is as follows:

```
import java.util.Scanner;
public class GCD_LCM {
static int gcd(int x, int y) {
int a, b;
a = (x > y) ? x : y; // a is greater number
b = (x < y) ? x : y; // b is smaller number
while (b != 0) { // Fixed the while loop condition
int temp = b;
b = a % b;
a = temp;
}
return a;
}
static int lcm(int x, int y) {
return (x * y) / gcd(x, y); // Calculate LCM using GCD
}
public static void main(String args[]) {
```

Scanner input = new Scanner(System.in);

System.out.println("Enter the two numbers: ");

int x = input.nextInt();

int y = input.nextInt();

System.out.println("The GCD of two numbers is: " + gcd(x, y));

System.out.println("The LCM of two numbers is: " + lcm(x, y));

input.close();

}

}

3) Knapsack

A. Program Inspection

1. The program contains an error in the following line: `int option1 = opt[n++][w];` The variable n is unintentionally incremented and should be: `int option1 = opt[n][w];`
2. The most effective category of program inspection for this code is Category C: Computation Errors, as the identified issue pertains to computation within loops.
3. Program inspection cannot identify runtime errors or logical errors that may occur during program execution.
4. The program inspection technique is valuable for detecting and resolving computation-related issues.

B. Debugging

1. To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.

3. The corrected executable code is as follows:

```
public class Knapsack {

public static void main(String[] args) {

int N = Integer.parseInt(args[0]); // number of items

int W = Integer.parseInt(args[1]); // maximum weight of knapsack

int[] profit = new int[N + 1];

int[] weight = new int[N + 1];

// Generate random instance, items 1..N

for (int n = 1; n <= N; n++) {

profit[n] = (int) (Math.random() * 1000);

weight[n] = (int) (Math.random() * W);

}

int[][] opt = new int[N + 1][W + 1];

boolean[][] sol = new boolean[N + 1][W + 1];

for (int n = 1; n <= N; n++) {

for (int w = 1; w <= W; w++) {

int option1 = opt[n - 1][w]; // Fixed the increment here

int option2 = Integer.MIN_VALUE;

if (weight[n] <= w)

option2 = profit[n] + opt[n - 1][w - weight[n]];

opt[n][w] = Math.max(option1, option2);

sol[n][w] = (option2 > option1);

}
```

}

// Rest of the code is fine

// Print results

System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" + "\t" + "Take");

for (int n = 1; n <= N; n++) {

System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);

}

}

}

4) Magic Number

A. Program Inspection

1. The program has two errors:
2. Error 1: In the inner while loop, the condition should be `while (sum > 0)` instead of `while (sum == 0)`.
3. Error 2: Inside the inner while loop, the lines are missing semicolons: `s = s * (sum / 10);` and `sum = sum % 10;` should be corrected as follows:
   ```
   s = s * (sum / 10);
   sum = sum % 10;
   ```
4. The most relevant category of program inspection for this code is Category C: Computation Errors, as it contains computation errors within the while loop.
5. Program inspection cannot identify runtime problems or logical errors that may occur during program execution.
6. The program inspection technique is beneficial for identifying and resolving computation-related issues.

B. Debugging

1. There are two errors in the program, as identified above.

2. To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of num and s during execution.

3. The corrected executable code is as follows:

```
import java.util.*;
public class MagicNumberCheck {
public static void main(String args[]) {
Scanner ob = new Scanner(System.in);
System.out.println("Enter the number to be checked.");
int n = ob.nextInt();
int sum = 0, num = n;
while (num > 9) {
sum = num;
int s = 0;
while (sum > 0) { // Fixed the condition here
s = s * (sum / 10);
sum = sum % 10; // Fixed the missing semicolon
}
num = s;
}
if (num == 1) {
System.out.println(n + " is a Magic Number.");
} else {
```

System.out.println(n + " is not a Magic Number.");

}

}

}

5 Merge Sort

A. Program Inspection

1. The program contains several errors:
2. Error 1: In the `mergeSort` method, the lines `int[] left = leftHalf(array + 1);` and `int[] right = rightHalf(array - 1);` need to be corrected. This appears to be an attempt to split the array, but it is not implemented correctly.
3. Error 2: The `leftHalf` and `rightHalf` methods are flawed. They should return the correct halves of the array.
4. Error 3: The `merge` method should accept the left and right arrays as inputs, rather than `left++` and `right--`.
5. The most relevant category of program inspection for this code is Category C: Computation Errors, as it contains computation-related issues.
6. Program inspection cannot identify runtime problems or logical errors that may occur during program execution.
7. The program inspection technique is valuable for identifying and addressing computation-related issues.

B. Debugging

1. There are multiple errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints to examine the values of left, right, and array during execution. You can also use breakpoints to check the values of i1 and i2 inside the merge method.

3. The corrected executable code is as follows:

```
import java.util.*;

public class MergeSort {

public static void main(String[] args) {

int[] list = {14, 32, 67, 76, 23, 41, 58, 85};

7

System.out.println("before: " + Arrays.toString(list));

mergeSort(list);

System.out.println("after: " + Arrays.toString(list));

}

public static void mergeSort(int[] array) {

if (array.length > 1) {

int[] left = leftHalf(array);

int[] right = rightHalf(array);

mergeSort(left);

mergeSort(right);

merge(array, left, right);

}

}

public static int[] leftHalf(int[] array) {

int size1 = array.length / 2;

int[] left = new int[size1];

for (int i = 0; i < size1; i++) {
```

```
left[i] = array[i];

}

return left;

}

public static int[] rightHalf(int[] array) {

int size1 = array.length / 2;

int size2 = array.length - size1;

int[] right = new int[size2];

for (int i = 0; i < size2; i++) {

right[i] = array[i + size1];

}

return right;

}

public static void merge(int[] result, int[] left, int[] right) {

int i1 = 0;

int i2 = 0;

for (int i = 0; i < result.length; i++) {

if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {

result[i] = left[i1];

i1++;

} else {

result[i] = right[i2];
i2++;

}

}

}

}
```

6) Multiply Matrices

A. Program Inspection

1. The program contains several errors:
2. Error 1: In the nested loops for matrix multiplication, the loop indices should start from 0 instead of -1.
3. Error 2: The error message when the matrix dimensions are incompatible should state, “Matrices with entered orders can’t be multiplied with each other,” instead of the incorrect wording.
4. The most relevant category of program inspection for this code is Category C: Computation Errors, as it contains computation-related issues.
5. Program inspection cannot identify runtime problems or logical errors that may occur during program execution.
6. The program inspection technique is valuable for identifying and resolving computation-related issues.

B. Debugging

1. There are multiple errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints to examine the values of c, d, k, and sum during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.

3. The corrected executable code is as follows:

```
import java.util.Scanner;

class MatrixMultiplication {

public static void main(String args[]) {

int m, n, p, q, sum = 0, c, d, k;

Scanner in = new Scanner(System.in);

System.out.println("Enter the number of rows and columns of the first matrix");

m = in.nextInt();

n = in.nextInt();

int first[][] = new int[m][n];

System.out.println("Enter the elements of the first matrix");

for (c = 0; c < m; c++)

for (d = 0; d < n; d++)

first[c][d] = in.nextInt();

System.out.println("Enter the number of rows and columns of the second matrix");

p = in.nextInt();

q = in.nextInt();

if (n != p)

System.out.println("Matrices with entered orders can't be multiplied

with each other.");

else {

int second[][] = new int[p][q];

int multiply[][] = new int[m][q];
}

System.out.println("Enter the elements of the second matrix");

for (c = 0; c < p; c++)

for (d = 0; d < q; d++)

second[c][d] = in.nextInt();

for (c = 0; c < m; c++) {

for (d = 0; d < q; d++) {

for (k = 0; k < p; k++) {

sum = sum + first[c][k] * second[k][d];

}

multiply[c][d] = sum;

sum = 0;

}

}

System.out.println("Product of entered matrices:-");

for (c = 0; c < m; c++) {

for (d = 0; d < q; d++)

System.out.print(multiply[c][d] + "\t");

System.out.print("\n");

}

}

}

}
```

7) Quadratic Probing

A. Program Inspection

1. The program contains multiple errors:
2. Error 1: The `insert` method has a typo in the line `i += (i + h / h−)`.
3. Error 2: In the `remove` method, there is a logic error in the loop for rehashing keys. It should be `i = (i + h * h++)`.
4. Error 3: In the `get` method, there is a logic error in the loop to locate the key. It should be `i = (i + h * h++)`.
5. The most relevant categories of program inspection for this code are Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax and semantic issues present.
6. The program inspection technique is valuable for identifying and correcting these errors, but it may not detect logical errors that affect the program's behavior.

B. Debugging

1. There are three errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints and step through the code while examining variables like i, h, tmp1, and tmp2. You should pay attention to the logic of the insert, remove, and get methods.

3. The corrected executable code is as follows:

```
import java.util.Scanner;

class QuadraticProbingHashTable {

private int currentSize, maxSize;

private String[] keys;

private String[] vals;

public QuadraticProbingHashTable(int capacity) {
```

```
currentSize = 0;

maxSize = capacity;

keys = new String[maxSize];

vals = new String[maxSize];

}

public void makeEmpty() {

currentSize = 0;

keys = new String[maxSize];

vals = a String[maxSize];

}

public int getSize() {

return currentSize;

}

public boolean isFull() {

return currentSize == maxSize;

}

public boolean isEmpty() {

return getSize() == 0;

}

public boolean contains(String key) {

return get(key) != null;

}
```

```
private int hash(String key) {

return key.hashCode() % maxSize;

}

public void insert(String key, String val) {

int tmp = hash(key);

int i = tmp, h = 1;

do {

if (keys[i] == null) {

keys[i] = key;

vals[i] = val;

currentSize++;

return;

}

if (keys[i].equals(key)) {

vals[i] = val;

return;

}

i += (h * h++) % maxSize;

} while (i != tmp);

}

public String get(String key) {

int i = hash(key), h = 1;
```

```
while (keys[i] != null) {

if (keys[i].equals(key))

return vals[i];

i = (i + h * h++) % maxSize;

}

return null;

}

public void remove(String key) {

if (!contains(key))

return;

int i = hash(key), h = 1;

while (!key.equals(keys[i]))

i = (i + h * h++) % maxSize;

keys[i] = vals[i] = null;

for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)

{

String tmp1 = keys[i], tmp2 = vals[i];

keys[i] = vals[i] = null;

currentSize--;

insert(tmp1, tmp2);

}

currentSize--;
```

```
}

public void printHashTable() {

System.out.println("\nHash Table: ");

for (int i = 0; i < maxSize; i++)

if (keys[i] != null)

System.out.println(keys[i] + " " + vals[i]);

System.out.println();

}

}

public class QuadraticProbingHashTableTest {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Hash Table Test\n\n");

System.println("Enter size");

QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

char ch;

do {

System.out.println("\nHash Table Operations\n");

System.out.println("1. insert");

System.out.println("2. remove");

System.out.println("3. get");

System.out.println("4. clear");
System.out.println("5. size");

int choice = scan.nextInt();

switch (choice) {

case 1:

System.out.println("Enter key and value");

qpht.insert(scan.next(), scan.next());

break;

case 2:

System.out.println("Enter key");

qpht.remove(scan.next());

break;

case 3:

System.out.println("Enter key");

System.out.println("Value = " + qpht.get(scan.next()));

break;

case 4:

qpht.makeEmpty();

System.out.println("Hash Table Cleared\n");

break;

case 5:

System.out.println("Size = " + qpht.getSize());

break;
default:

System.out.println("Wrong Entry\n");

break;

}

qpht.printHashTable();

System.out.println("\nDo you want to continue (Type y or n) \n");

ch = scan.next().charAt(0);

13

} while (ch == 'Y' || ch == 'y');

}

}
```

8) Sorting Array

A. Program Inspection

1. Errors identified:
2. Error 1: The class name “Ascending Order” contains an extra space and an underscore. It should be corrected to “AscendingOrder.”
3. Error 2: The first nested for loop has an incorrect loop condition `for (int i = 0; i ¿= n; i++);` which should be modified to `for (int i = 0; i < n; i++).`
4. Error 3: There is an unnecessary semicolon (;) after the first nested for loop that should be removed.
5. The most relevant categories of program inspection are Category A: Syntax Errors and Category B: Semantic Errors, as the code contains both types of issues.
6. Program inspection can identify and correct syntax errors and some semantic issues; however, it may not detect logic errors that impact the program's behavior.
7. The program inspection technique is beneficial for fixing syntax and semantic errors, but debugging is necessary to address logic errors.

B. Debugging

1. There are two errors in the program as identified above.

2. To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.

3. The corrected executable code is as follows:

```
import java.util.Scanner;

public class AscendingOrder {

public static void main(String[] args) {

int n, temp;

Scanner s = new Scanner(System.in);

System.out.print("Enter the number of elements you want in the array: ");

n = s.nextInt();

int a[] = new int[n];

System.out.println("Enter all the elements:");

for (int i = 0; i < n; i++) {

a[i] = s.nextInt();

}

for (int i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

if (a[i] > a[j]) {

temp = a[i];

a[i] = a[j];
a[j] = temp;

}

}

}

System.out.print("Ascending Order: ");

for (int i = 0; i < n - 1; i++) {

System.out.print(a[i] + ", ");

}

System.out.print(a[n - 1]);

}

}
```

9 Stack Implementation

A. Program Inspection

1. Errors identified:
2. Error 1: The `push` method uses a decrement operation on the `top` variable (`top–`) instead of an increment operation. It should be corrected to `top++` to push values correctly.
3. Error 2: The `display` method has an incorrect loop condition in `for(int i=0; i ¿ top; i++).` The loop condition should be `for (int i = 0; i <= top; i++)` to correctly display the elements.
4. Error 3: The `pop` method is missing in the `StackMethods` class. It should be added to provide a complete stack implementation.
5. The most relevant category of program inspection is Category A: Syntax Errors, as there are syntax issues in the code. Additionally, Category B: Semantic Errors can help identify logic and functionality problems.

6. The program inspection technique is valuable for identifying and fixing syntax errors, but further inspection is necessary to ensure the logic and functionality are correct.

B. Debugging

1. There are three errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.

3. The corrected executable code is as follows:

```
public class StackMethods {
private int top;
int size;
int[] stack;
public StackMethods(int arraySize) {
size = arraySize;
stack = new int[size];
top = -1;
}

public void push(int value) {
if (top == size - 1) {
System.out.println("Stack is full, can't push a value");
} else {
```

```
top++;

stack[top] = value;

}

}

public void pop() {

if (!isEmpty()) {

top--;

} else {

System.out.println("Can't pop...stack is empty");

}

}

public boolean isEmpty() {

return top == -1;

}

public void display() {

for (int i = 0; i <= top; i++) {

System.out.print(stack[i] + " ");

}

System.out.println();

}

}
```

10) Tower of Hanoi

A. Program Inspection

1. Errors identified:
2. Error 1: In the line `doTowers(topN ++, inter−, from+1, to+1)`, the increment and decrement operators are used incorrectly. It should be corrected to `doTowers(topN - 1, inter, from, to)`.
3. The most relevant category of program inspection is Category B: Semantic Errors, as the issues in the code pertain to logic and functionality.
4. The program inspection technique is valuable for identifying and fixing semantic errors in the code.

.

B. Debugging

1. There is one error in the program, as identified above.

2. To fix this error, you need to replace the line: doTowers(topN ++, inter--, from+1, to+1);

3. with the correct version: doTowers(topN - 1, inter, from, to);

4. The corrected executable code is as follows:

```
public class MainClass {

public static void main(String[] args) {

int nDisks = 3;

doTowers(nDisks, 'A', 'B', 'C');

}

public static void doTowers(int topN, char from, char inter, char to) {

if (topN == 1) {

System.out.println("Disk 1 from " + from + " to " + to);

} else {

doTowers(topN - 1, from, to, inter);
System.out.println("Disk " + topN + " from " + from + " to " + to);

doTowers(topN - 1, inter, from, to);

}

}

}
```

III. Static Analysis Tools:

(Used cppcheck for static analysis)

In text format:

```
\tokenizer.h:12:0: information: Include file: <iostream> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <iostream>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:13:0: information: Include file: <string> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <string>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:14:0: information: Include file: <vector> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <vector>

^
```

```
C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:15:0: information: Include file: <fstream> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <fstream>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:16:0: information: Include file: <sstream> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <sstream>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:17:0: information: Include file: <algorithm> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <algorithm>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\scanner.h:12:0: information: Include file: <fstream> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <fstream>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\scanner.h:13:0: information: Include file: <string> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <string>

^
```

```
C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:12:0: information: Include file: <iostream> not found. Please note: Cppcheck does not need standard library headers to get proper results. [missingIncludeSystem]

#include <iostream>

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\tokenizer.h:27:1: style: The class 'Token' does not declare a constructor although it has private member variables which likely require initialization. [noConstructor]

class Token

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parsetreeleaf.h:14:1: style: The class 'ParseTreeLeaf' does not declare a constructor although it has private member variables which likely require initialization. [noConstructor]

class ParseTreeLeaf {

^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:15:9: warning: Member variable 'Parser::parser' is not initialized in the constructor. [uninitMemberVar]

Parser::Parser()

        ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:15:9: warning: Member variable 'Parser::symbolList' is not initialized in the constructor. [uninitMemberVar]

Parser::Parser()

        ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:15:9: warning: Member variable 'Parser::numSymbols' is not initialized in the constructor. [uninitMemberVar]

Parser::Parser()
```

```
        ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:15:9: warning: Member
variable 'Parser::symbolIndex' is not initialized in the constructor. [uninitMemberVar]

Parser::Parser()

        ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:15:9: warning: Member
variable 'Parser::tree' is not initialized in the constructor. [uninitMemberVar]

Parser::Parser()

        ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\scanner.h:19:2: style: Class 'Scanner'
has a constructor with 1 argument that is not explicit. [noExplicitConstructor]

 Scanner(std::string filename);

 ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:499:36: style: Condition
'!(*parser).get().token==LEFT_PAREN' is always false [knownConditionTrueFalse]

    else if(!(*parser).get().token == LEFT_PAREN)

                                   ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:636:38: performance:
Function parameter 'filename' should be passed by const reference. [passedByValue]

ParseTree* Parser::start(std::string filename)

                                     ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:661:31: performance:
Function parameter 's' should be passed by const reference. [passedByValue]

bool Parser::isin(std::string s) {

                              ^
```

```
C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:638:18: performance: Passing the result of c_str() to a function that takes std::string as argument no. 1 is slow and redundant. [stlcstrParam]

    parser = new Scanner(filename.c_str());

                 ^

C:\Users\202201466\Desktop\[202201466_Lab3_2.c:10]\parser.cpp:636:0: style: The function 'start' is never used. [unusedFunction]

ParseTree* Parser::start(std::string filename)

^

nofile:0:0: information: Active checkers: 167/835 (use --checkers-report=<filename> to see details) [checkersReport]
```