

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Vítor Tiago Varajão Martins

Detection of Plagiarism in Software in an Academic Environment

Master dissertation

Supervised by: Pedro Rangel Henriques

Daniela da Cruz

Braga, June 16, 2015

ACKNOWLEDGEMENTS

We give thanks to Dr. Jurriaan Hage for the copy of the Marble tool and Bob Zeidman for the CodeSuite licenses.

This work is funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology), within project PEst-OE/EEI/UI0752/2014.

ABSTRACT

We illustrate the state of the art in software plagiarism detection tools by comparing their features and testing them against a wide range of source codes. The source codes are copies of the same file disguised to hide plagiarism and show the tools accuracy at detecting each type.

The decision to focus our research on plagiarism of programming languages is two fold: on one hand, it is a challenging case-study since programming languages impose a structured writing style; on the other hand, we are looking for the integration of such a tool in an Automatic-Grading System (AGS) developed to support teachers in the context of *Programming courses*.

Based on that analysis, we set out to develop our own tool and describe the process from the architecture to the implementation. With examples of how the algorithms works. Followed by the analysis of the finished tool in terms of speed and accuracy against the ones in existence.

RESUMO

Nós ilustramos o estado da arte no que toca a ferramentas de detecção de plágio de software comparando as ferramentas existentes em termos de funcionalidades e testando-as contra um leque de códigos fonte. Os códigos são cópias do mesmo ficheiro disfarçados de forma a esconder o plágio e permitem-nos analisar a eficácia das ferramentas na detecção de cada um dos tipos.

A decisão de focar a pesquisa no plágio de linguagens de programação teve dois objectivos: por um lado, é um caso de estudo desafiante já que as linguagens de programação requerem uma escrita estruturada; por outro lado, estamos à procura de integrar uma ferramenta num Sistema de Avaliação-Automática (AGS) desenvolvido para ajudar os professores no contexto de *Disciplinas de programação*.

Com base nessa análise, prosseguimos com a construção da nossa ferramenta e descrevemos o processo desde a arquitectura à implementação. Com exemplos de como funcionam os algoritmos. Finalizando com a análise da ferramenta acabada em termos de velocidade e eficácia contra as existentes.

CONTENTS

Contents iii

1	INTRODUCTION	3
1.1	Objectives	4
1.2	Research Hypothesis	4
1.3	Document structure	4
2	STATE OF THE ART	5
2.1	Approaches	6
2.2	Existing tools	7
2.2.1	CodeMatch	7
2.2.2	CPD	8
2.2.3	JPlag	8
2.2.4	Marble	8
2.2.5	MOSS	8
2.2.6	Plaggie	9
2.2.7	SIM	9
2.2.8	Sydney's Sherlock	9
2.2.9	Warwick's Sherlock	10
2.2.10	YAP	10
2.3	Comparison of the existing tools	11
2.3.1	Timeline	11
2.3.2	Feature comparison	11
2.3.3	Comparison of the results	12
2.3.4	Strategy for testing the tools	14
2.3.5	Testing the tools	14
3	SPECTOR ARCHITECTURE	17
3.1	Requirements	17
3.2	Abstract Syntax Trees	18
3.2.1	Artificial Nodes	21
3.2.2	Identifiers	21
3.2.3	Expressions	22
3.2.4	Control structures	22
3.2.5	Blocks	22
4	SPECTOR IMPLEMENTATION	23

Contents

4.1	Development Decisions	23
4.2	Features	25
4.3	Implementation Structure	26
4.4	Methodology	26
4.4.1	Algorithm 1: Unaltered copy	27
4.4.2	Comments changed	29
4.4.3	Algorithm 2: Identifiers changed	29
4.4.4	Scope changed	34
4.4.5	Algorithm 3: Operands order switched	34
4.4.6	Algorithm 4: Variable types and control structures replaced	41
4.4.7	Statements order switched	46
4.4.8	Algorithm 5: Group of calls turned into a function call or vice versa	46
4.4.9	Main Algorithm	51
5	SPECTOR TESTS	53
6	CONCLUSIONS AND FUTURE WORK	56
A	SOURCE CODES USED IN THE TESTS	61
A.1	Hightlights	61
A.2	The program, Calculator	61
A.2.1	Original source code for the Calculator exercise and the 1st type of plagiarism.	62
A.2.2	Source code for the Calculator exercise, with the 2nd type of plagiarism.	63
A.2.3	Source code for the Calculator exercise, with the 3rd type of plagiarism.	65
A.2.4	Source code for the Calculator exercise, with the 4th type of plagiarism.	67
A.2.5	Source code for the Calculator exercise, with the 5th type of plagiarism.	69
A.2.6	Source code for the Calculator exercise, with the 6th type of plagiarism.	71
A.2.7	Source code for the Calculator exercise, with the 7th type of plagiarism.	72
A.2.8	Source code for the Calculator exercise, with the 8th type of plagiarism.	74
A.3	The program, 21 Matches	76
A.3.1	Original source code for the 21 Matches exercise and the 1st type of plagiarism.	76
A.3.2	Source code for the 21 Matches exercise, with the 2nd type of plagiarism.	80
A.3.3	Source code for the 21 Matches exercise, with the 3rd type of plagiarism.	84
A.3.4	Source code for the 21 Matches exercise, with the 4th type of plagiarism.	88
A.3.5	Source code for the 21 Matches exercise, with the 5th type of plagiarism.	92
A.3.6	Source code for the 21 Matches exercise, with the 6th type of plagiarism.	95
A.3.7	Source code for the 21 Matches exercise, with the 7th type of plagiarism.	99
A.3.8	Source code for the 21 Matches exercise, with the 8th type of plagiarism.	103
B	TEST RESULTS	108

Contents

B.1	CodeMatch results	108
	B.1.1 Results for the Calculator source codes	108
	B.1.2 Results for the 21 Matches source codes	109
B.2	JPlag results	109
	B.2.1 Results for the Calculator source codes	110
	B.2.2 Results for the 21 Matches source codes	110
B.3	Marble results	111
	B.3.1 Results for the Calculator source codes	111
B.4	MOSS results	112
	B.4.1 Results for the Calculator source codes	112
	B.4.2 Results for the 21 Matches source codes	112
B.5	SIM results	113
	B.5.1 Results for the Calculator source codes	113
	B.5.2 Results for the 21 Matches source codes	114
B.6	Sydney's Sherlock results	114
	B.6.1 Results for the Calculator source codes	115
	B.6.2 Results for the 21 Matches source codes	116
B.7	Warwick's Sherlock results	117
	B.7.1 Results for the Calculator source codes	117
	B.7.2 Results for the 21 Matches source codes	119
B.8	Spector results	120
	B.8.1 Results for the Calculator source codes	120

LIST OF FIGURES

Figure 1	A timeline showing the years in which each tool was developed or referenced.	11
Figure 2	An overview of the results obtained for the Calculator, Java source codes.	13
Figure 3	The structure for our application.	17
Figure 4	A tree similar to the parse tree produced from the Greet source code.	20
Figure 5	The AST for the source code.	21
Figure 6	A block diagram representing Spector's implementation structure.	26
Figure 7	AST generated from source code 1A.	28
Figure 8	AST generated from source code 1B.	28
Figure 9	AST generated from source code 2A.	30
Figure 10	AST generated from source code 2B.	30
Figure 11	AST generated from source code 2A.	33
Figure 12	AST generated from source code 2C.	33
Figure 13	AST generated from source code 3A.	36
Figure 14	AST generated from source code 3B.	36
Figure 15	AST generated from source code 3A.	39
Figure 16	AST generated from source code 3C.	39
Figure 17	AST generated from source code 4A.	42
Figure 18	AST generated from source code 4B.	42
Figure 19	AST generated from source code 4C.	44
Figure 20	AST generated from source code 4D.	44
Figure 21	Source codes 5A and 5B, respectively.	47
Figure 22	AST generated from source code 5A.	48
Figure 23	AST generated from source code 5B.	48
Figure 24	AST generated from source code 5C.	50
Figure 25	AST generated from source code 5D.	50
Figure 26	An overview of the results obtained for the Calculator, Java source codes.	53
Figure 27	Part of the HTML presenting the results for the Calculator, Java source codes.	54
Figure 28	Part of the HTML presenting the results for the 21 Matches, Java source codes.	55

LIST OF TABLES

Table 1	Comparison of the plagiarism detection tools.	12
Table 2	The results from the CodeMatch tool for the Calculator source codes	109
Table 3	The results from the CodeMatch tool for the 21 Matches source codes	109
Table 4	The results from the JPlag tool for the Calculator source codes	110
Table 5	The results from the JPlag tool for the 21 Matches source codes	111
Table 6	The results from the Marble tool for the Calculator source codes	111
Table 7	The results from the MOSS tool for the Calculator source codes	112
Table 8	The results from the MOSS tool for the 21 Matches source codes	113
Table 9	The results from the SIM tool for the Calculator source codes	114
Table 10	The results from the SIM tool for the 21 Matches source codes	114
Table 11	The results from the Sherlock tool for the Calculator source codes	115
Table 12	The results produced by Sherlock tool for the Calculator source codes with the -n 2 argument	115
Table 13	The results produced by Sherlock tool for the Calculator source codes with the -z 3 argument	116
Table 14	The results from the Sherlock tool for the 21 Matches source codes	117
Table 15	The results from Warwick's Sherlock tool for the Calculator source codes, using the No Comments + Normalized transformation	118
Table 16	The results from the Warwick's Sherlock tool for the Calculator source codes, using the Tokenized transformation	118
Table 17	The results from Warwick's Sherlock tool for the 21 Matches source codes, using the No Comments + Normalized transformation	119
Table 18	The results from the Warwick's Sherlock tool for the 21 Matches source codes, using the Tokenized transformation	119
Table 19	The results from the Spector tool for the Calculator source codes	120

LIST OF LISTINGS

3.1	The source code for the Greet example.	19
4.1	Source code 1A.	27
4.2	Source code 1B.	27
4.3	Source code 2A.	30
4.4	Source code 2B.	30
4.5	Source code 2A.	32
4.6	Source code 2C.	32
4.7	Source code 3A.	35
4.8	Source code 3B.	35
4.9	Source code 3A.	38
4.10	Source code 3C.	38
4.11	Source code 4A.	42
4.12	Source code 4B.	42
4.13	Source code 4C.	44
4.14	Source code 4D.	44
4.15	Source code 5A.	47
4.16	Source code 5B.	47
4.17	Source code 5C.	49
4.18	Source code 5D.	49
A.1	Original source code for the Calculator exercise and the 1st type of plagiarism.	62
A.2	Source code for the Calculator exercise with the 2nd type of plagiarism.	63
A.3	Source code for the Calculator exercise with the 3rd type of plagiarism.	65
A.4	Source code for the Calculator exercise with the 4th type of plagiarism.	67
A.5	Source code for the Calculator exercise with the 5th type of plagiarism.	69
A.6	Source code for the Calculator exercise with the 6th type of plagiarism.	71
A.7	Source code for the Calculator exercise with the 7th type of plagiarism.	72
A.8	Source code for the Calculator exercise with the 8th type of plagiarism.	74
A.9	Original source code for the 21 Matches exercise and the 1st type of plagiarism.	77
A.10	Source code for the 21 Matches exercise with the 2nd type of plagiarism.	80
A.11	Source code for the 21 Matches exercise with the 3rd type of plagiarism.	84
A.12	Source code for the 21 Matches exercise with the 4th type of plagiarism.	88

List of Listings

A.13 Source code for the 21 Matches exercise with the 5th type of plagiarism.	92
A.14 Source code for the 21 Matches exercise with the 6th type of plagiarism.	95
A.15 Source code for the 21 Matches exercise with the 7th type of plagiarism.	99
A.16 Source code for the 21 Matches exercise with the 8th type of plagiarism.	103

INTRODUCTION

Nowadays it is very easy to copy documents. If the source of a copy, be it partial or wholesome, is wrongfully identified then it is plagiarism. Even when the original document was authored by the same person (in which case it is copying).

In an academic environment, if a student plagiarizes, a teacher will be unable to properly grade his ability. This is a problem that applies not only to text documents but to source-code as well. This is why there is a need to recognize plagiarism. However, with a large number of documents, this burdensome task should be computer aided.

There are many programs for the detection of source-code plagiarism. Some of them are offline tools, like the **Sherlock** tools (Joy and Luck, 1999; Hage et al., 2010), **YAP**¹ (Wise, 1996; Li and Zhong, 2010; Hage et al., 2010), **Plaggie** (Ahtiainen et al., 2006; Hage et al., 2010), **SIM**² (Grune and Huntjens, 1989; Ahtiainen et al., 2006; Hage et al., 2010), **Marble** (Hage et al., 2010), and **CPD**³ (Copeland, 2003) which, even though it was only made to detect copies, is still a useful tool. There are also online tools like **JPlag** (Prechelt et al., 2000; Li and Zhong, 2010; Cui et al., 2010; Ahtiainen et al., 2006; Hage et al., 2010; Liu et al., 2006) and **MOSS** (Schleimer, 2003; Li and Zhong, 2010; Cui et al., 2010; Ahtiainen et al., 2006; Hage et al., 2010; Liu et al., 2006), and even tool sets like **CodeSuite** (Shay et al., 2010).

Due to the complexity of the problem itself, it is often hard to create software that accurately detects plagiarism, since there are many ways a programmer can alter a program without changing its functionality.

As we stated previously, there are two Sherlock tools, one from the University of Sydney and another from the University of Warwick so we will distinguish them as Sydney's Sherlock and Warwick's Sherlock, respectively.

Along the last years, those alterations were collected, studied and classified. At least three types were identified: lexical changes; structural changes; and technological changes.

Many of the tools (like JPlag or MOSS) transform the programs into tokens and compare them, thus being ineffective at detecting structural changes (Li and Zhong, 2010; Cui et al., 2010). This is why the approach based on the usage of an abstraction, such as an *Abstract Syntax Tree (AST)*, was proven

1 Yet Another Plague

2 software and text SIMilarity tester

3 Copy-Paste-Detector

1.1. Objectives

to be better at identifying a much larger scope of plagiarism cases (Parker and Hamblen, 1989; Faidhi and Robinson, 1987).

There is a need for the production of a tool that can compare the structure of two programs in order to accurately identify plagiarism, with the motivation of making it easy to extend to other languages and use in other systems, such as Quimera (Fonte et al., 2012), an *Automatic Grading System*.

1.1 OBJECTIVES

Our purpose is to study the state-of-the-art of this area and analyze the advantages and disadvantages of the existing methodologies.

The aim is to produce an improved system that will use *ASTs* as an abstraction, in order to detect source-code plagiarism based on the structures of input programs.

The tool will be created in the *Java* language and, at a first stage, it will support *Java*. For its development, *ANTLR*⁴ (Parr and Quong, 1995) will be used to generate the *front-end* that reads the input programs and produces the *ASTs*. Using this approach, it will be easy to extend the tool to support other languages (creating additional *front-ends*).

The tool will be subjected to a variety of tests to analyze its performance in comparison to similar tools. These tests will be done with programs developed by ourselves and with programs developed by students.

1.2 RESEARCH HYPOTHESIS

To reach our objectives (see Section 1.1) we will study the use of the *AST* trees and define procedures to compare those trees, according to our specifications. As seen by Baxter et al. (1998); Cui et al. (2010); Li and Zhong (2010), the trees can be used as an effective method to detect copies and plagiarism, which reinforces our hypothesis.

1.3 DOCUMENT STRUCTURE

This document is organized using the following structure: Chapter 1 introduces the problem, context and motivation, and our objectives (see Section 1.1). Chapter 2 presents the state of the art in the detection of software plagiarism; It also includes several tests (see Section 2.3.5) which were made with source code that was built using specific types of plagiarism — this will show the quality of the results obtained with the state of the art tools. Chapters 3 and 4 detail the process behind the development of Spector, an *AST* based tool, leading to its architecture and implementation. Chapter 5 shows a comparison between the results from our tool and the previous ones, along further test cases.

⁴ ANother Tool for Language Recognition

1.3. Document structure

Finally, Chapter 6 will close with the conclusions reached through this study and the possibilities for future research and development.

STATE OF THE ART

There are several techniques for the detection of plagiarism in source code. Their objective is to stop unwarranted plagiarism of source code in academic and commercial environments.

If a student uses existing code, it must be in conformance with the teacher and the school rules. The student might have to build software from the ground up, instead of using existing source code or tools that could greatly reduce the effort required to produce it but, this will allow a teacher to properly grade the student according to his knowledge and effort.

If a company uses existing source code, it may be breaking copyright laws and be subjected to lawsuits because it does not have its owners consent.

This need led to the development of plagiarism detectors, this is, programs that take text files (natural language documents, programs or program components) as input and output a percentage showing their probability of being copies of each other.

To understand the functionalities in these tools, it is necessary to understand the terms used to describe them.

The following is a list presenting such terms:

TOKEN A word or group of characters, sometimes named *n-gram* where *n* denotes the number of characters per token. Since white-spaces and newlines are usually ignored, the input "while(true) {" could produce two 5-grams: "while(" and "true){".

TOKENIZATION The conversion of a natural text into a group of tokens, as shown in the previous definition (Token).

HASH A number computed from a sequence of characters, usually used to speed up comparisons. If we use the ASCII¹ values of each character, we could turn the token "word" into 444 (119 + 111 + 114 + 100).

HASHING The conversion of a sequence of tokens into their respective hash numbers.

FINGERPRINT A group of characteristics that identify a program, much like physical fingerprints are used to identify people. An example would be if we consider a fingerprint to be composed by 3 hashes, the sum of the differences (between hashes) to be the algorithm used for matching

¹ American Standard Code for Information Interchange

2.1. Approaches

them and a value of 10 to be the threshold. In which case, taking a pair of files with the fingerprints: [41,582,493] and [40,585,490]. The program would match, as the sum of the differences is 9 ($|41 - 40| + |582 - 585| + |493 - 490| = 1 + 3 + 3$).

STRUCTURAL INFORMATION This is information from the structure of a programming language. An example would be the concept of comments, conditional controls, among others.

The implementations of these concepts is always specific to each tool and; since this process is not usually explained, it can only be inspected by analyzing its source code.

2.1 APPROACHES

In general, this type of programs will usually build some form of fingerprint for each file, as to improve the efficiency of the comparisons. From our research, we saw that the following methodologies were used to produce the fingerprints.

- An attribute-based methodology, where metrics are computed from the source code and used for the comparison. A simple example would be: using the size of the source code (number of characters, words and lines) as an attribute to single out the source codes that had a very different size. This methodology was mentioned by [Wise \(1992\)](#).
- A token-based methodology, where the source code is tokenized and hashed, creating a fingerprint for each file or method. The fingerprints are then matched using hash comparison algorithms and accepted when within a threshold. This methodology was used by [Wise \(1993\)](#) and by [Schleimer \(2003\)](#).
- A structure-based methodology, where the source code is abstracted to an Internal Representation (IR), like an AST² or a PDG³. The IRs are then used to compare the source codes, allowing for an accurate comparison. This methodology was used by [Baxter et al. \(1998\)](#) and by [Li and Zhong \(2010\)](#).

These methodologies go from the least accuracy and the highest efficiency to the highest accuracy and the lowest efficiency.

Some examples of the metrics that could be used in an abstract-based methodology would be: files size, number of variables, number of functions and number of classes, among others. These metrics will usually be insufficient as students will usually solve the same exercise, which would cause a lot of suspicion from the tool.

² Abstract Syntax Tree

³ Program Dependency Graph

2.2. Existing tools

The token-based methodology came with an attempt to balance the accuracy and the efficiency, often using RKS-GST⁴ Wise (1993) which is a modern tokenization and hashing algorithm. To improve the results even further, some tools mix some structure dependent metrics and modifications such as, removing comments (used by JPlag 2.2.3) or sorting the order of the functions (used by Marble 2.2.4).

The structure-based approach uses abstractions that maintain the structure of the source code. This makes the technique more dependent on the language but it will also make the detection immune to several types of plagiarism such as, switching identifier names, switching the positions of functions and others.

2.2 EXISTING TOOLS

We found several tools for the detection of software plagiarism throughout our research. Some of those tools were downloaded, used for testing purposes and thus can be discussed hereafter. Other tools, like Plague (Whale, 1990; Wise, 1996; Joy and Luck, 1999) and GPLag (Liu et al., 2006; Bejarano et al., 2013) were not accounted for. Each of the tools analyzed is presented here, ordered alphabetically. The following criteria, inspired on Hage et al. (2010) were used to compare the tools:

1ST) SUPPORTED LANGUAGES: The languages supported by the tool.

2ND) EXTENDABILITY: Whether an effort was made to make adding languages easier.

3RD) QUALITY OF THE RESULTS: If the results are descriptive enough to distinguish plagiary from false positives.

4TH) INTERFACE: If the tool has a GUI⁵ or presents its results in a graphical manner.

5TH) EXCLUSION OF CODE: Whether the tool can ignore base code.

6TH) SUBMISSION AS GROUPS OF FILES: If the tool can consider a group of files as a submission.

7TH) LOCAL: If the tool can work without needing access to an external web service.

8TH) OPEN SOURCE: If the source code was released under an open source license.

2.2.1 CodeMatch

CodeMatch (Hage et al., 2010) is a part of CodeSuite (Shay et al., 2010) and detects plagiarism in source code by using algorithms to match statements, strings, instruction sequences and identi-

4 Running Karp-Rabin matching and Greedy String Tiling

5 Graphical User Interface

2.2. Existing tools

fiers. CodeSuite is a commercial tool that was made by SAFE⁶, which is housed at <http://www.safe-corp.biz/index.htm>. It features several tools to measure and analyze source or executable code.

This tool is only available as an executable file (binary file) and only runs under Windows.

CodeMatch supports the following languages:

- ABAP, ASM-6502, ASM-65C02, ASM-65816, ASM-M68k, BASIC, C, C++, C#, COBOL, Delphi, Flash ActionScript, Fortran, FoxPro, Go, Java, JavaScript, LISP, LotusScript, MASM, MATLAB, Pascal, Perl, PHP, PL/M, PowerBuilder, Prolog, Python, RealBasic, Ruby, Scala, SQL, TCL, Verilog, VHDL, Visual Basic

2.2.2 CPD

CPD⁷ (Copeland, 2003) is a similarity detector that is part of PMD, a source code analyzer that finds inefficient or redundant code, and is housed at <http://pmd.sourceforge.net/>. It uses the RKS-GST⁸ (Wise, 1993) algorithm to find similar code.

It supports the following languages:

- C, C++, C#, Java, EcmaScript, Fortran, Java, JSP, PHP, Ruby

2.2.3 JPlag

JPlag (Prechelt et al., 2000; Li and Zhong, 2010; Cui et al., 2010; Ahtiainen et al., 2006; Hage et al., 2010; Liu et al., 2006) takes the language structure into consideration, as opposed to just comparing the bytes in the files. This makes it good for detecting plagiarism despite the attempts of disguising it. It supports the following languages:

- C, C++, C#, Java, Scheme, Natural language

2.2.4 Marble

Marble (Hage et al., 2010), which is described in <http://www.cs.uu.nl/research/techreps/aut/jur.html>, is a modern tool (made in 2010) for plagiarism detection. It facilitates the addition of languages by using code normalizers to make tokens that are independent of the language. A RKS algorithm is then used to detect similarity among those tokens.

It supports the following languages:

- Java, C#, Perl, PHP, XSLT

6 Software Analysis and Forensic Engineering

7 Copy-Paste-Detector

8 Running Karp-Rabin matching and Greedy String Tiling

2.2. Existing tools

2.2.5 MOSS

MOSS (Schleimer, 2003; Li and Zhong, 2010; Cui et al., 2010; Ahtiainen et al., 2006; Hage et al., 2010; Liu et al., 2006) automatically detects similarity between programs with a main focus on detecting plagiarism in several languages that are used in programming classes. It is provided as an Internet service that presents the results in HTML⁹ pages, reporting the similarities found, as well as the code responsible. It can ignore base code that was provided to the students and focuses on discarding information while retaining the critical parts in order to avoid false positives.

Its HTML interface was produced by the author of JPlag (Guido Malpohl) and, the results come in both an overview and detailed forms.

It supports the following languages:

- A8086 Assembly, Ada, C, C++, C#, Fortran, Haskell, HCL2, Java, Javascript, Lisp, Matlab, ML, MIPS Assembly, Modula2, Pascal, Perl, Python, Scheme, Spice, TCL, Verilog, VHDL, Visual Basic

2.2.6 Plaggie

Plaggie (Ahtiainen et al., 2006; Hage et al., 2010), which is housed at <http://www.cs.hut.fi/Software/Plaggie/>, detects plagiarism in Java programming exercises. It was made for an academic environment where there was a need to ignore base code.

It supports the following languages:

- Java

2.2.7 SIM

SIM¹⁰ (Grune and Huntjens, 1989; Ahtiainen et al., 2006; Hage et al., 2010), which is housed at http://dickgrune.com/Programs/similarity_tester/, is an efficient plagiarism detection tool which has a command line interface, like the Sherlock tool. It uses a custom algorithm to find the longest common substring, which is order-insensitive.

It supports the following languages:

- C, Java, Pascal, Modula-2, Lisp, Miranda, Natural language

⁹ Hyper Text Markup Language

¹⁰ software and text SIMilarity tester

2.2. Existing tools

2.2.8 Sydney's Sherlock

The University of Sydney's Sherlock (Hage et al., 2010), which is housed at <http://sydney.edu.au/engineering/it/~scilect/sherlock/>, detects plagiarism in documents through the comparison of fingerprints which, as stated in the website, are a sequence of *digital signatures*. Those digital signatures are simply a hash of (3, by default) words.

It allows for control over the threshold, the number of words per *digital signature* and the *granularity* of the comparison (which is 4, by default) by use of the respective arguments: -t, -n and -z.

It supports the following languages:

- Natural language

2.2.9 Warwick's Sherlock

The University of Warwick's Sherlock (Joy and Luck, 1999), which is housed at <http://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/> provides a GUI which allows the user to choose what tests should be used (compare normalized code, ignore comments, etc) as well as change the tests inner thresholds. The results are shown through a connection graph, statistical results and the sums of the individual percentual results. One can also access the detailed results and see a visual representation of the connections between both files contents.

It supports the following languages:

- Natural language, Java and C++

2.2.10 YAP

YAP ¹¹ (Wise, 1996; Li and Zhong, 2010; Hage et al., 2010), which is housed at <http://www.pam1.bcs.uwa.edu.au/~michaelw/YAP.html>, is a tool that currently has 3 implementations, each using a fingerprinting methodology with different algorithms. The implementations have a tokenizing and a similarity checking phase and just change the second phase.

The tool itself is an extension of Plague.

YAP1 The initial implementation was done as a Bourne-shell script and uses a lot of shell utilities such as diff, thus being inefficient. It was presented by Wise (Wise (1992)).

YAP2 The second implementation was made as a Perl script and uses an external C implementation of the Heckel (Heckel, 1978) algorithm.

¹¹ Yet Another Plague

2.3. Comparison of the existing tools

YAP3 The third and latest iteration uses the RKS-GST methodology.

The tools themselves are separate from the tokenizers but packaged together. These tools are said to be viable at the detection of plagiarism on natural language, aside from their supported languages.

It supports the following languages:

- Pascal, C and LISP

2.3 COMPARISON OF THE EXISTING TOOLS

From the information obtained and the results computed, we have produced a variety of comparisons that highlight the tools accuracy and behavior as well as a view on the progress done in the area of plagiarism detection in source code.

2.3.1 *Timeline*

A timeline was produced (see Figure 1), indicating the years when the tools were developed or, at least mentioned in an article. This shows us the progression of source code plagiarism detection tools.

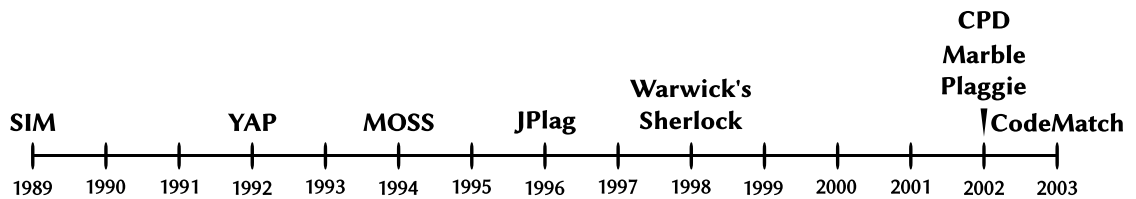


Figure 1.: A timeline showing the years in which each tool was developed or referenced.

Note that we omitted Sydney's Sherlock since we could not ascertain the date of its creation.

2.3.2 *Feature comparison*

A table (see Table 1) was also produced to report the criteria defined previously (see Section 2.2) for each tool. The values can be ✓(Yes), ✗(No), a ? on the cases where we could not ascertain if the feature is present or a number, in the case of supported languages.

2.3. Comparison of the existing tools

Name	1	2	3	4	5	6	7	8
CodeMatch	36	✗	✓	✓	✓	✗	✓	✗
CPD	6	✓	✗	✓	?	?	✓	✓
JPlag	6	✗	✓	✓	✓	✓	✗	✗
Marble	5	✓	✓	✗	✓	?	✓	✗
MOSS	25	✗	✓	✓	✓	✓	✗	✗
Plaggie	1	?	?	✓	?	?	✓	✓
SIM	7	?	✓	✗	✗	✗	✓	?
Sydney's Sherlock	1	✗	✗	✗	✗	✗	✓	?
Warwick's Sherlock	3	✓	✓	✓	✓	?	✓	✓
YAP	5	?	✓	✗	✓	?	✓	✗

Table 1.: Comparison of the plagiarism detection tools.

We can observe that both the CodeMatch and the MOSS tools support several languages, which makes them the best choices when analyzing languages that the other tools do not support. However, others like CPD or Marble are easily extendable to cope with more languages. Note that if the tools support *natural language*, they can detect plagiarism between any text document but will not take advantage of any structural information.

Overall, we found that GUIs are unnecessary to give detailed output, so long as the tool can produce descriptive results that indicate the exact lines of code that caused the suspicion. This is observable with the use of Marble, SIM and YAP tools as they do not offer GUIs but still have descriptive results. On the other hand, tools like Sydney's Sherlock do not present enough output information.

On academic environments, both the 5th and the 6th criteria are very important as they allow teachers to filter unwanted source code from being used in the matches. This means that tools like JPlag and MOSS allow for the proper filtering of the input source code.

As said in the introduction, the 7th criteria reveals that JPlag and MOSS are the only tools dependent on online services as they are web tools.

In what concerns the availability of tools on open licenses, only CPD and Plaggie satisfy that requirement. For those wanting to reuse or adapt the tools, the 7th and the 8th criteria are important as the tool would need to have a license allowing its free use, and integration would benefit from having the tool distributed alongside the application.

2.3.3 Comparison of the results

Figure 2 sums up the results that were obtained by comparing the Calculator Java source codes (as detailed in Section s:tests) and gives us an overview. Each number represents the type of plagiarism that was used to hide the plagiarism on the files that were compared.

2.3. Comparison of the existing tools

We can see that the 1st type of plagiarism (exact copy) was easily detected by all the tools but the other types of plagiarism always had a big impact on some of the tools. Here are the most noticeable:

- MOSS has the most trouble with a lot of the types of plagiarism as it tries very hard to avoid false-positives, thus discarding a lot of information.
- Sydney's Sherlock has a lot of trouble with the 2nd type of plagiarism (comments changed) as it compares the entire source codes without ignoring the comments.
- CodeMatch has the most trouble with the 3rd type of plagiarism (identifiers changed) as its algorithms must make some sort of match between identifiers.

Note that we used the results from Warwick's Sherlock using the "no comments, normalized" transformation as it gave us a full table.

We can see that most tools have a hard time when comparing from the 6th to the 8th types of plagiarism as they had a lot of small changes or movements of several blocks of source code. These are the types of plagiarism that would be best detected with the use of a structural methodology as, despite those changes to the structure the context remains intact.

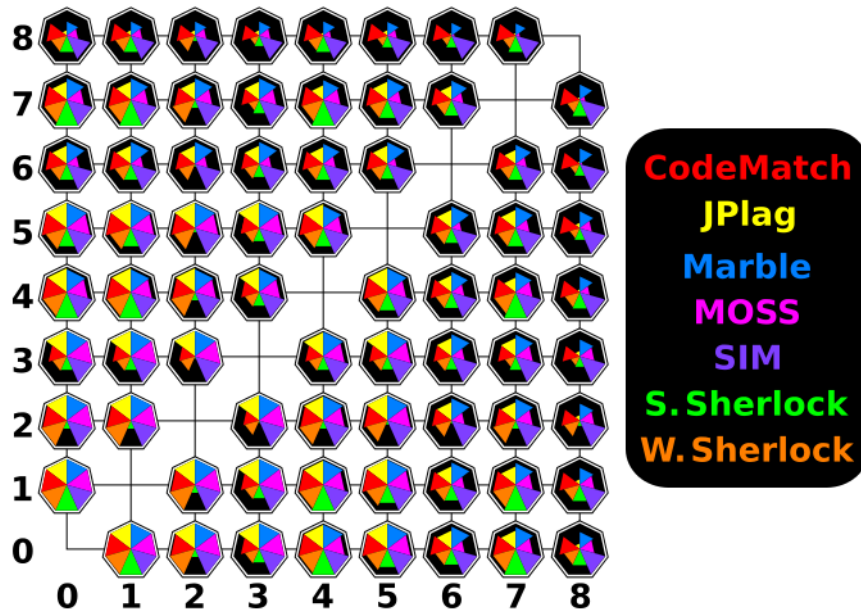


Figure 2.: An overview of the results obtained for the Calculator, Java source codes.

We can conclude that every tool has a weakness when it comes to certain types of plagiarism but notice that none of the results are 0%, the triangle representing some matches are just very small and easy to miss. We also confirm that no type of plagiarism can fool all the tools at the same time.

2.3. Comparison of the existing tools

2.3.4 *Strategy for testing the tools*

Because plagiarism starts from an original file, we can construct a list of types of plagiarism showing how complex the changes were. This allows us to measure just how versed a plagiarist is in the language. List 2.3.4 details each type of plagiarism, ordered by complexity. This list was based on those found in Joy and Luck (1999); Bejarano et al. (2013); Cosma and Joy (2008).

- Lexical changes
 1. Unaltered copy.
 2. Comments changed.
 3. Identifiers changed.
 4. Scope changed (making a local variable into a global one).
- Structural changes
 5. Operands order switched (e.g. $x < y$ to $y \geq x$).
 6. Variable types and control structures replaced with equivalents.
 7. Statements order switched.
 8. Group of calls turned into a function call or vice versa.

The two types of plagiarism listed below were also referred in the lists. However, we will not consider them as they depend entirely on the teachers supervision since they are hard to identify.

- Generating source-code by use of code-generating software.
- Making a program from an existing program in a different language.

Based on these types of plagiarism, a few exercises were collected and edited. These exercises are all cases of plagiarism based on an original source code and will demonstrate the accuracy of the existing tools for each type, provided that they support the language used.

The source codes of every exercise can be found in Appendix A, along with a description indicating their intended functionality.

2.3.5 *Testing the tools*

As described (see Section 2.3.4), 8 types of plagiarism were considered and a pair of source files (see Appendix A) were created with an original file and 8 files, modified according to the type of plagiarism.

2.3. Comparison of the existing tools

1ST TYPE OF PLAGIARISM This type of plagiarism is an exact copy of the original. It is the simplest type of plagiarism as no modifications were done to hide it.

2ND TYPE OF PLAGIARISM This type of plagiarism is when the comments are changed or removed. It is the easiest modification as it has no risk of affecting the code.

Note that most plagiarism detectors either ignore comments or have an option to ignore them and will not be diverted by this type of plagiarism. Of course, to do so the tools need the syntactic information on how comments are defined in the language.

3RD TYPE OF PLAGIARISM This type of plagiarism is done by changing every identifier such as variable or function names.

4TH TYPE OF PLAGIARISM This type of plagiarism is when we turn local variables into global ones, and vice versa.

5TH TYPE OF PLAGIARISM This type of plagiarism is when the operands in comparisons and mathematical operations are changed (e.g. $x < y$ to $y \geq x$).

6TH TYPE OF PLAGIARISM This type of plagiarism is when variable types and control structures are replaced with equivalents.

Care has to be taken as to avoid breaking the codes functionality since the types will need to be converted to have the same behavior.

7TH TYPE OF PLAGIARISM This type of plagiarism is when the order of statements (read lines) is switched.

This is a common type of plagiarism as it only takes care to make sure that the source code will show the same behavior.

8TH TYPE OF PLAGIARISM This type of plagiarism is when groups of calls are turned into a function call or vice versa.

To demonstrate the accuracy of the existing tools detecting the different cases of theft, two sample programs were collected and edited based on the types of plagiarism listed above. As the selected tools are prepared to work with Java or C programs, the exercises were written in both languages.

The following list of actions were performed to create the eight variants:

2.3. Comparison of the existing tools

- to produce the first case, it is a straightforward file copy operation.
- to apply the second strategy, we simply removed the comments or changed their contents.
- to create a third variant, we changed most variable and function identifiers into reasonable replacements.
- to produce a copy of the fourth type, we moved a group of local variables into a global scope and removed them from the function arguments. The type declaration was removed but the initialization was kept.
- to obtain a file for the fifth type, we switched the orders of several comparisons and attributions. One of those operations was to replace $x + = y - 5$ by $x + = -5 + y$, which is a clear attempt at hiding plagiarism.
- to get a file for the sixth type, we replaced variable types such as *int* and *char* with *float* and *string*, along with the necessary modifications to have them work the same way.
- to create another copy according to the seventh type, we moved several statements, even some which broke the behavior (write after read) to consider cases where the plagiarist was not careful.
- to produce a file exhibiting the eighth type, we applied this type of plagiarism in the easy (move entire functions) and the hard (move specific blocks of code in to a function) ways.

All of the results obtained can be found in Appendix B.

The results are displayed in terms of percentage, representing the level of similarity. The results and respective tables can be found at: <http://www4.di.uminho.pt/~gepl/Spector/paper/>. To compare any result tables, we used the following algorithm: Given two tables, we subtract the second to the first parts and add their results. As an example, consider the tables [2,1] and [2,3]. In this case the formula is: $(2-2)+(3-1)$, which is equal to 2. A positive value indicates that the matches were overall higher. We will refer to this metric as the difference metric (DM). When there are several lines of results, their average is calculated by dividing the sum of each line by the number of lines.

Since most tools only give a percentage for each pair of files, most tables are symmetrical. For the asymmetrical tables, the percentage must be read as the percentage of the line file that matches the column file. As an example, if we look at Table 7, it should be read that 11% of the L6.c (6) file matches the L8.c (8) file and that 8% of the L8.c file matches the L6.c file.

SPECTOR ARCHITECTURE

As stated in Chapter 1, the objective of the work proposed is to create an application that will use *ASTs* as an abstraction to detect plagiarism. This tool was named Spector (**S**ource **I**nspector).

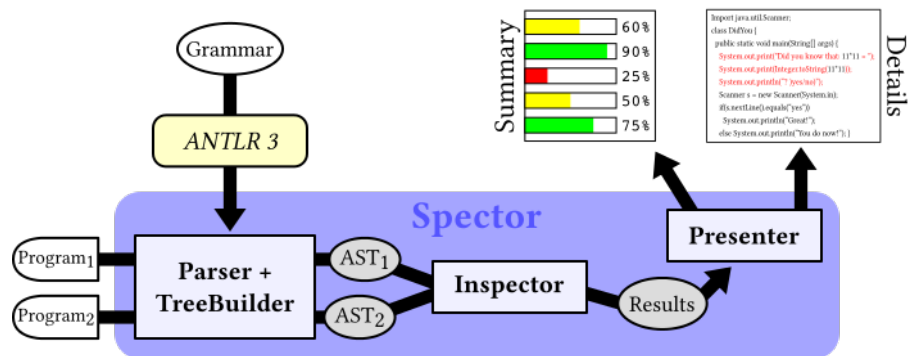


Figure 3.: The structure for our application.

First of all, let us refer to a Grammar as G_L , a Parser+TreeBuilder as TB_L and each input Program as a P_L . L is the programming language which they use or process.

In Figure 3 we can observe that given a G_A , we can produce a TB_A using *ANTLR 3*. This TB_A can then be used to translate a P_A into an *AST*.

Given a pair of Programs (let us say $P_{A,1}$ and $P_{A,2}$), a TB_A will be used to produce AST_1 and AST_2 . These *ASTs* will then be delivered to a Manager, this will send them to an Inspector and will then send the output metrics to a Presenter.

All the information related to an input Source Code can be grouped into a *Suspect*, namely its *AST* and its origin (the *File*). So we can say that an Inspector produces Metrics based on pairs of Suspects.

3.1 REQUIREMENTS

Since the program is going to detect plagiarism, we can look at the types of plagiarism (in Section 2.3.4) as target cases.

This creates the need to identify:

- Comments (an annotation that is ignored),

3.2. Abstract Syntax Trees

- **Identifiers** (the names of classes, variables, etc),
- Scopes (if a variable was defined inside or outside a method),
- **Expression elements** (for ex.: $x, >, 1$),
- Types (int, float, etc),
- **Control structures** (if, while, etc),
- Statements (an action, for ex.: print "Hello"),
- **Blocks** (a group of statements enclosed between `{}`).

We decided to focus on the items in bold as the others are unnecessary to detect similarity. Take Identifiers for example, if we want to find similar Identifiers we do not need to check their scope or type, we can simply see if they are used in the same places and have similar behaviors. Note that Blocks usually follow a class or method declaration (for ex.: "void main() ...") and Statements are inside such Blocks, even in the cases of single line Blocks where the `{` and `}` characters are optional.

3.2 ABSTRACT SYNTAX TREES

As stated, we transform each source code into an Abstract Syntax Tree (*AST*) and use it to measure the similarity between two source codes.

When the parser extracts the tokens (symbols) from a source code, it produces a parse tree that is then converted into an *AST*. The nodes of the *AST* used in this project contain information such as: the position of the originating text; its contents and its type.

To be more concrete, a node is an object of the `CommonTree` class which has the following variables:

- Token token
- int startIndex
- int stopIndex
- CommonTree parent
- int childIndex

This class is mostly a `Token` along with the attributes required to build a tree. The start and stop indexes have the character position from the start of the token to the end of its children and the `childIndex` is the position of this node in its parents list of children.

Since the class extends a `BaseTree`, it gets a list of children, along with the methods to manipulate it and a few useful methods, namely:

3.2. Abstract Syntax Trees

GETFIRSTCHILDWITHTYPE Returns the first child with a specific type.

REPLACECHILDREN Replaces every child between two indexes with a new child.

GETANCESTORS Returns a list with the nodes from the root to this nodes parent.

HASANCESTOR Checks whether there is an ancestor with a specific type.

GETANCESTOR Gets the first ancestor with a specific type.

TOSTRINGTREE Returns a string representation of the entire tree.

The Token class itself has the following variables:

- int type
- String text
- int line
- int start
- int stop

The first 3 are the tokens: type, contents and line position. The start and stop variables have the character position for the start and the end of this token.

Looking at the information provided by this class, we can see that traversing, accessing and manipulating a tree composed by these nodes is not a big issue. However, by controlling the transformation from a parse tree into an AST, we can remove unnecessary nodes and manipulate its structure.

As an example, let us consider the source code in Listing 3.1.

```
1 package Greetings;
2 import static java.lang.System.out;
3
4 public class Greet {
5     public static String hello = "Hello World";
6
7     // This function adds two values
8     public static int sum(int a, int b) {
9         return a+b;
10    }
11
12    public static int main(String[] args) {
13        int a = 2;
14
15        if(a == 2) {
16            return 2;
17        }
18    }
19 }
```

3.2. Abstract Syntax Trees

```

18
19     switch(a) {
20         case 3:
21             out.println(message);
22             break;
23     }
24 }
25 }

```

Listing 3.1: The source code for the Greet example.

This source code contains all of the relevant information that we want to detect (seen in the previous list of requirements).

When the parser processes that source code, it will produce a parse tree that shows the node types, similar to Fig. 4.

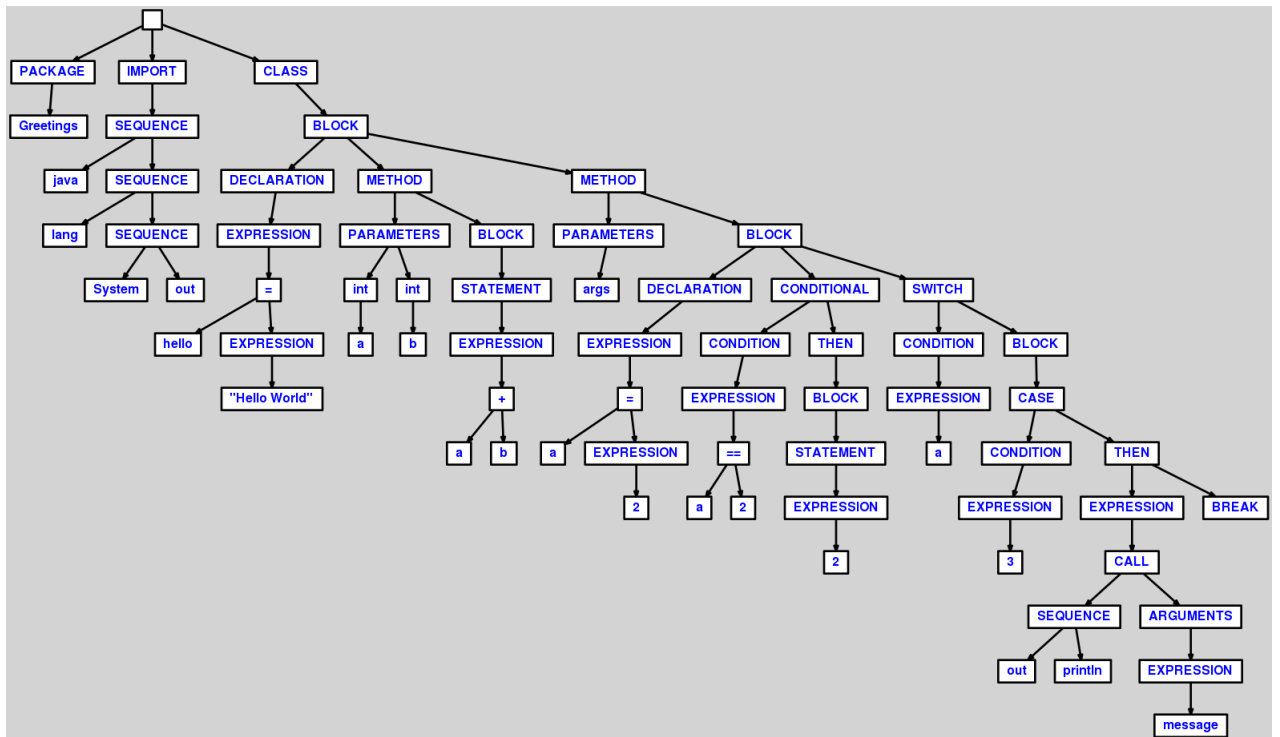


Figure 4.: A tree similar to the parse tree produced from the Greet source code.

Note that the actual parse tree contains every single node that was detected and we only show the relevant nodes and types here. This tree shows us all of the types underneath the nodes, with the exception of primitive nodes like identifiers and literals.

3.2. Abstract Syntax Trees

When this tree is transformed into an *AST*, our rules determine which nodes are relevant and how they are structured. Using these rules, AntLR generated the *AST* in Fig. 5.

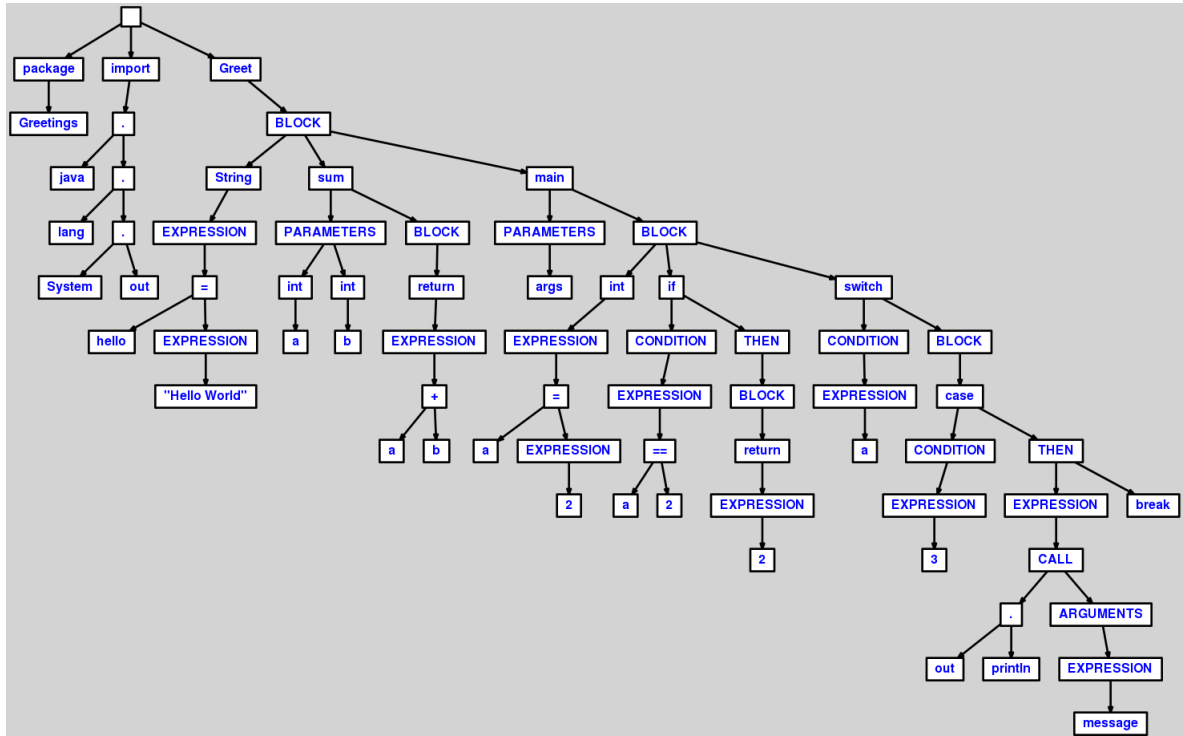


Figure 5.: The *AST* for the source code.

As we can see, it is easy to figure out how the source code works just by looking at this *AST*. So long as we can acquire and structure the relevant information, detecting similarity is a matter of choosing what to look for and how to process that information.

Let us now look at the decisions that influenced how certain types of nodes would appear in the *AST*.

3.2.1 Artificial Nodes

To make the *AST* easier to understand, we added artificial nodes like "CALL" and "ARGUMENTS" which tell us what kind of object that part of the *AST* refers.

3.2.2 Identifiers

We can see that some of the nodes were merged with their identifiers, as seen in the case of the CLASS and METHOD nodes. This allows these nodes to have the identifiers content while retaining

3.2. Abstract Syntax Trees

their type. This means that instead of a CLASS node with an IDENTIFIER as a child node, we get a CLASS node with "Greet" as its contents. This also means that we need to consider several types when looking for identifiers, which is why we created the TokenGroup class. This class enumerates types such as "Identifier" that are associated with a group of AntLR tokens, by a Nexus. This allows us to specify that CLASS and METHOD nodes are also identifiers, in the sense that they have that type of content.

While it is easier to read merged nodes, one must notice that some nodes could not do this as their identifier may be a SEQUENCE. Meaning that it is a group of identifiers, separated by "." characters. Examples can be seen by looking at the IMPORT and CASE nodes which have the identifier as their first child node.

3.2.3 Expressions

Since expressions are mostly composed by literals we decided to structure them in the following form: The parent node is either a literal or an operator, in which case it is followed by one or two children. This makes the order of operators such as "+" and "*" irrelevant as they will always become the parent.

3.2.4 Control structures

As we can see, a CONDITIONAL is a control structure like an "if" or a "while" which contains a CONDITION. Since they can be given the same functionality, we consider them to be the same type. The only exception is the "case" conditional since it has an intrinsic "==" in its condition, along with the contents of the parent "switch" condition.

A TokenType class was built which, similar to the TokenGroup class, enumerates types that are assigned to a token that was generated by AntLR. This allows us to add specific nodes to a CommonTree, allowing for the proper comparison between a case and another conditional.

3.2.5 Blocks

Blocks are the containers of statements and can be found in relation to some nodes like the class and method types. It is important to identify blocks as they represent the objects functionality which is a crucial detail whenever we want to see if two objects are the same.

SPECTOR IMPLEMENTATION

Given our past experience, we chose to use *ANTLR 3* in conjunction with the *Java* programming language. This would allow us to produce *ASTs* by getting a grammar for the target language and adding rewrite rules. As an added benefit, we could use the *ANTLR Works* tool to generate and view *ASTs* while editing the grammar.

We started by obtaining a few *Java* grammars from <http://www.antlr3.org/grammar/list>. There are several grammars available, however some led to dead links, and others are for earlier versions of *ANTLR* or for earlier versions of *Java*. After a trial period, we settled with the "Java 1.6" grammar (dated Jan 16, 2009) as the one that was best suited to our needs. Then, we used *ANTLR Works* to add rewrite rules as well as to build and view the produced *ASTs*.

Due to the language complexity, the grammar was quite big (over 2500 lines). This made testing and viewing the produced *ASTs* with *ANTLR Works* a slow and error prone task. To make matters worse, there were exceptions that were reported for unexpected reasons. Even if we were using the same test and did not change anything before or after the occurrence.

Of course, given the aforementioned problems, we decided to bypass *ANTLR Works* and just use *ANTLR 3.5.2* directly from the command line. A decision that greatly improved the speed of development but had us lose the visual *ASTs*. This was later resolved when we were able to use the *ANTLR DOTTreeGenerator* along with the *StringTemplate* classes to produce DOT files within *Spector*. This allowed us to once again have visual representations of the resulting *ASTs* without needing to use *ANTLR Works* and be subjected to its quirks. We then produced an *AST* that followed our structure (shown in Section 3.2).

4.1 DEVELOPMENT DECISIONS

To keep *Spector* modular, we decided to split its functionality into the following packages and respective classes/enumerations:

- lang

4.1. Development Decisions

NEXUS An abstract class that will be extended for each language. These classes are responsible for interfacing with the *Parser+TreeBuilders* and produce *ASTs/DOTs* from source code files.

SUSPECT This class represents one of the files to be compared and has its *AST* along with other informations.

TOKENGROUP An enumeration of types that are associated with one or more AntLR tokens, to facilitate their location.

TOKENTYPE This is another enumeration whose types are associated with single tokens. They are used to add artificial tokens.

- **spector**

FILEHANDLER A class that handles input from files and/or folders.

SPECTOR The main program responsible for receiving input and managing the other classes.

INSPECTOR A class that has the methods for calculating similarity measures and comparing Suspects.

COMPARISON The class that holds a pair of Suspects and the informations calculated by an Inspector.

PRESENTER A class that generates presentations from a list of Comparisons.

Initially there was only the Spector class along with the Parser+TreeBuilder files, which were generated from a grammar. As this was not modular or easy to manage, we created the FileHandler and Nexus classes. At this point Spector could produce *ASTs* from combinations of files and/or folders. We then grouped all of the relevant information into a single Suspect class that was used by an Inspector class to compare them. This led to the creation of the TokenGroup and TokenType classes: two enumerations of internal groups of types and single types, respectively. Both of these are mapped with AntLR tokens by each Nexus, thus facilitating the location and addition of tokens to the *ASTs*. We finally added the Candidate and Presenter classes which respectively contain a pair of Suspects and results and present the results (as an example: by generating HTML files).

Note that the lang folder also contains folders named after the language, these have the necessary parsers and lexers, as well as its Nexus. As an example: The Java language has a lang.java package which contains:

JAVA.G The grammar used to generate the parser and lexer files.

JAVA.TOKENS A file defining every AntLR token in the grammar.

JVALEXER.JAVA The lexer that gathers the tokens.

JAVAPARSER.JAVA A parser that produces and restructures the *ASTs*.

4.2. Features

JAVANEXUS.JAVA The nexus that handles the communication with these files.

With a structure designed following these decisions we could add a language by:

1. Creating or obtaining a grammar for the desired language,
2. Add the necessary rewrite rules to produce the *AST*,
3. Build a class that extends Nexus and implements its abstract methods,
4. Add the associations between AntLR Tokens, TokenGroups and TokenTypees.

While not entirely simple, one can look at the existing Java grammar and JavaNexus classes as examples which, combined with *ANTLR Works*, can speed the development up.

4.2 FEATURES

Given the list of criteria that was used to compare the plagiarism detectors (in Section 2.2), we can pinpoint the most important features (in bold) for use within Academic Environments:

1. Supported languages
2. Extendability
3. **Quality of the results**
4. Interface
5. **Exclusion of code**
6. **Submission as groups of files**
7. **Local**
8. Open source

We decided that those 4 features selected (in bold) should constitute the essential requirements for the basic application. The remaining features, not crucial, may be implemented at a later time.

To implement the 5th and 6th features, the application should accept the following types of input:

- Many files (f?) : fA, fB, fC
- One directory (d?) with files : dX/fA, dX/fB, dX/fC
- Many directories (each a submission) : dX/fA, dY/fA, dZ/fA

4.3. Implementation Structure

- One or more directories (each an exercise) with subdirectories (each a submission) : e1/dX/fA, e1/dY/fA, e2/dX/fA, e2/dY/fA

With this we have enough to build an application but we still need a Structure that brings it all together.

4.3 IMPLEMENTATION STRUCTURE

We can see *Spector's* structure in Fig. 6. It will have a *CLI*¹ that will implement all of its functionalities. A *GUI* is also planned as a future motivation, to show its ease of integration. We can also see that within *Spector* there is a *Nexus* class that interacts with a *Parser+TreeBuilder* and produces *ASTs*; Along with an *Inspector* class which produces measures.

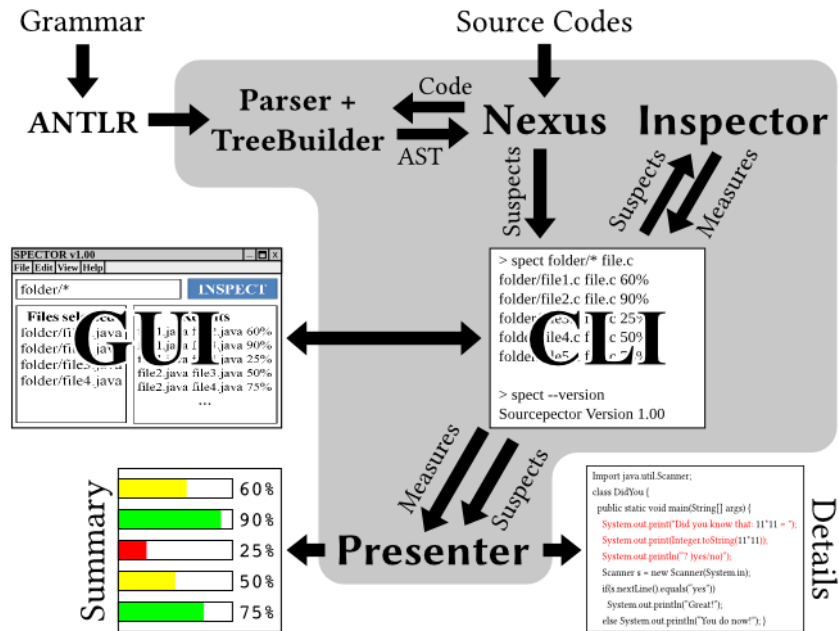


Figure 6.: A block diagram representing Spector's implementation structure.

4.4 METHODOLOGY

The actual plagiarism detection process will follow a methodology which comprises an algorithm for each type of plagiarism (see Section 2.3.4) and a main algorithm. Each algorithm takes two Suspects and returns a percentile measure which indicates their similarity.

Before we start, note that we will be using "{ }" characters to represent maps and "[]" for arrays. We also use the "()" characters when we want to group several objects but this should be seen as "for every one of these items".

¹ Command Line Interface

4.4. Methodology

4.4.1 Algorithm 1: Unaltered copy

Starting with the first type of plagiarism, we need to match the contents of every pair of nodes. As this is a heavy comparison, we first make sure that the *ASTs* are similar enough by using intermediate metrics. Our proposed algorithm goes as follows:

1. If the number of nodes is equal,
2. And the number of types of nodes is equal,
3. And all node contents are equal,
 - a) (Then) Return 100%.
4. (Otherwise) Return 0%.

As we can see, this algorithm uses 3 metrics with increasing strictness to determine the similarity between two source codes. As an example of its functionality, let us consider source codes 1A and 1B (in Listings 4.1 and 4.2).

```
1 public class Example {
2     public void main() {
3         int x = 0;
4         int y = x;
5         return 369;
6     }
7 }
```

Listing 4.1: Source code 1A.

```
1 public class Example {
2     public void main() {
3         int x = 0;
4         return 369;
5         int y = x;
6     }
7 }
```

Listing 4.2: Source code 1B.

As we know, these will in turn generate the *ASTs* that are used in the comparison. Looking at those *ASTs* (in Figs. 7 and 8), we can see that they represent the functionality of the previous source codes accurately.

4.4. Methodology

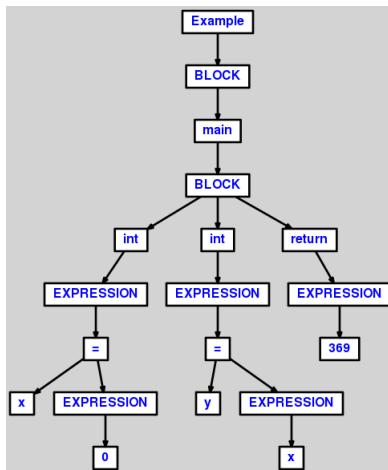


Figure 7.: AST generated from source code 1A.

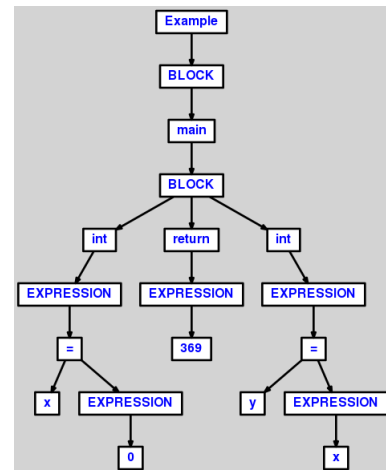


Figure 8.: AST generated from source code 1B.

Now, imagine we are comparing 1A to itself. The algorithm would start by calculating the number of nodes in each source code (19). It would then calculate the number of nodes by type:

- 1 *class* node (Example),
- 2 *block* nodes,
- 1 *method* node (main),
- 2 *type* nodes (int),
- 1 *return* node,
- 5 *expression* nodes,
- 2 *assignment* nodes (=),
- 5 *identifier* nodes (x, y, x),
- 2 *primitive* nodes (0, 369).

And finally, it would check if the contents are equal by comparing each pair of nodes. Since every test would be successful, the algorithm would end by returning a measure of 100%. Which means that these ASTs are considered exact copies of each other.

If we compare ASTs 1A to 1B, we can see that the algorithm will go through every step but will fail in the last one since all node contents must be equal, resulting in a measure of 0%.

Note that this algorithm is the only one that either returns 100% or 0%, meaning that it does not account for other cases; its measure must not affect the final measure unless it is 100%.

4.4. Methodology

4.4.2 *Comments changed*

As Comments are not part of our targets (see Section 3.1) we can simply ignore this type of plagiarism.

4.4.3 *Algorithm 2: Identifiers changed*

From this point onwards we will consider that there is a threshold that defines the strictness of the comparisons. As an example: If we were comparing the number of nodes between two *ASTs* and the first had 10 nodes, a threshold of 20% means that the second *AST* needs to have between 8 and 12 nodes to be considered similar.

1. Map each identifier name to its occurrence nodes (IOM_1, IOM_2),
2. Calculate the highest number of identifiers between the IOMs (A),
3. For each pair of identifier names between the IOMs,
 - a) If their number of occurrences is similar,
 - i. Add the pair to a map (Candidates),
4. Calculate the number of identifiers in the Candidates map (B),
5. Measure += $(B / A) * 0.18$,
6. For each pair of identifier names in the Candidates map,
 - a) If their number of occurrences by category² is similar,
 - i. Add the pair to a map (Suspects),
7. Calculate the number of identifiers in the Suspects map (C),
8. Measure += $(C / B) * 0.42$,
9. For each pair of identifier names in the Suspects map,
 - a) If they have a similar behavior³,
 - i. Add the pair to a map (Equivalences),
10. Calculate the number of identifiers in the Equivalences map (D),
11. Measure += $(D / C) * 0.38$,
12. For each pair of identifier names in the Equivalences map,

² In grammatical terms, these categories are the terminals that were assigned to integers by ANTLR (in a .tokens file)

³ The behavior of a pair of identifiers is considered similar if their parent nodes have the same category and the neighboring nodes which are not identifiers have the same contents.

4.4. Methodology

- a) If their contents are the same,
 - i. Add the pair to a map (Copies),
- 13. Calculate the number of identifiers in the Copies map (E),
- 14. Measure += $(E / D) * 0.02$,
- 15. Return Measure.

Note that Identifier-Occurrence Maps (IOM) are built. These Maps associate each identifier names to all the nodes that use it afterwards. Those maps are then merged into a map of Candidates which associates the nodes in both IOMs that are similar. From here, a new map is built from the previous and a metric is applied to further filter the possible suspects until we obtain a map of exact copies.

As an example of the algorithms functionality, let us consider source codes 2A and 2B (see Listings 4.3 and 4.4).

```

1 public class Example {
2   public void main() {
3     int x = 7;
4     int y = x*3;
5     y = -y;
6     return y;
7   }
8 }

```

Listing 4.3: Source code 2A.

```

1 public class Example {
2   public void main() {
3     int x = 7;
4     int y = 3*x;
5     y = -y;
6     return y;
7   }
8 }

```

Listing 4.4: Source code 2B.

Looking at the ASTs that were generated from the previous source codes (in Figs. 9 and 10) we get a clear look at the input that the algorithm receives.

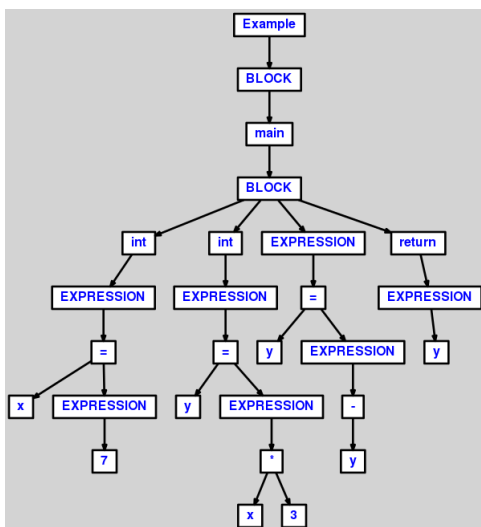


Figure 9.: AST generated from source code 2A.

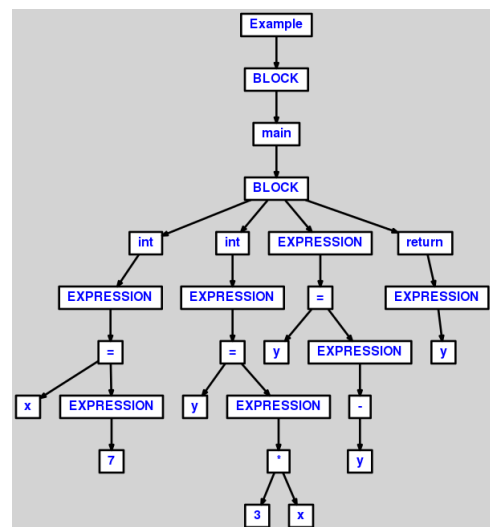


Figure 10.: AST generated from source code 2B.

4.4. Methodology

If it were to compare source codes 2A to 2B, the algorithm would start with the creation of the IOMs:

- IOM_{2A}: "Example":[1], "main":[2], "x":[3,4], "y":[4,5,5,6],
- IOM_{2B}: "Example":[1], "main":[2], "x":[3,4], "y":[4,5,5,6].

From here, it would calculate the higher number of identifiers between the IOMs ($A = 4$). The algorithm would then check if each pair of identifiers had the same number of occurrences and adds it to a map:

Candidates: {

- "Example":["Example", "main"],
- "main":["Example", "main"],
- "x":["x"],
- "y":["y"].

}

The number of identifiers in that map ($B = 4$) would then be used to calculate the first addition to the measure: $\text{Measure} = (B / A = 4 / 4 = 1) * 0.18 = 0.18$,

Traversing the map of Candidates, it would then calculate the number of types of occurrence for each entry and add the pairs with an equal number to a map:

Suspects {

- "Example":["Example"],
- "main":["main"],
- "x":["x"],
- "y":["y"].

}

Once again, it will calculate the number of identifiers in this map ($C = 4$) and use that value to add to the measure: $\text{Measure} += (C / B = 4 / 4 = 1) * 0.42 = 0.42$.

Then, looking at the map of Suspects, it would check each node to its occurrences and add the ones with similar behaviors to a map:

Equivalences {

- "Example":["Example"],
- "main":["main"],
- "x":["x"],

4.4. Methodology

- "y":["y"].

}

It will calculate the number of identifiers in this map ($D = 4$) and use that value to add to the measure:

Measure += $(D / C = 4 / 4 = 1) * 0.38 = 0.38$.

Finally, the map of Equivalent nodes will be filtered according to its contents and a map will retain the pairs that are exact copies:

Copies {

- "Example":["Example"],
- "main":["main"],
- "x":["x"],
- "y":["y"].

}

Followed by calculating its number of identifiers ($E = 4$) and making a final addition to the measure:

Measure += $(E / D = 4 / 4 = 1) * 0.02 = 0.02$,

This gives a resulting measure of $0.18 + 0.42 + 0.38 + 0.02 = 1.0$ (100%). Meaning that the source codes are exact copies in terms of identifiers.

As we can see, this strategy is built by comparing both *ASTs* with metrics that get more detailed as they add to the measure.

Let us now consider the comparison between source codes 2A and the new 2C (see Listings 4.5 and 4.6).

```
1 public class Example {
2     public void main() {
3         int x = 7;
4         int y = x*3;
5         y = -y;
6         return y;
7     }
8 }
```

Listing 4.5: Source code 2A.

```
1 public class Copy {
2     public void main() {
3         int a = 7;
4         int b = a+1;
5         int c = a*3;
6         return c;
7     }
8 }
```

Listing 4.6: Source code 2C.

Looking at these source codes, we can see that: The x and y variables were changed with a and c , respectively; the $b=a+1$ node was added and the " $y=-y$ " node was removed. This shows us that the similarity between these source codes will be harder to find.

If we look at the *ASTs* that were generated (in Figs. 11 and 12), we can see that the structure changed quite a bit.

4.4. Methodology

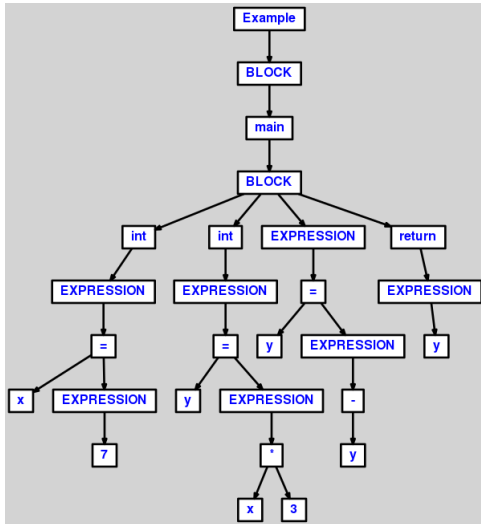


Figure 11.: AST generated from source code 2A.

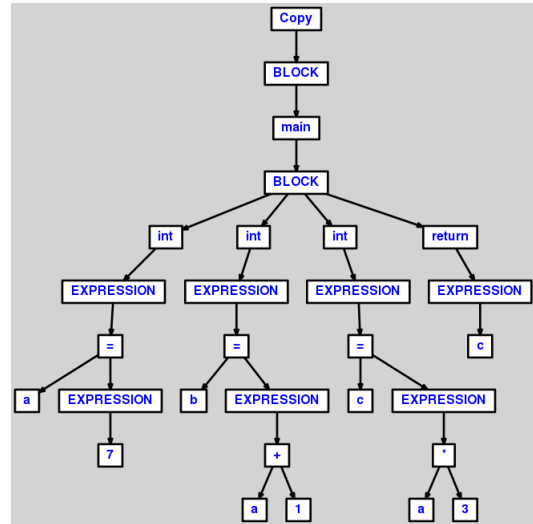


Figure 12.: AST generated from source code 2C.

Once again, we start by building the *IOMs*:

- IOM_{2A} : "Example":[1], "main":[2], "x":[3,4], "y":[4,5,5,6]
- IOM_{2C} : "Copy":[1], "main":[2], "a":[3,4,5], "b":[4], "c":[5,6]

Then we calculate the higher number of identifiers in the *IOMs* ($A = 5$).

Once again, we compare the number of occurrences and produce a map from the pairs of nodes that are similar:

Candidates {

- ("Example", "main"):["Copy", "main", "b"],
- "x":["c"],

}

Note that due to the changes, some of the identifiers were not associated. This could still be changed by setting a threshold bigger than the default (0.0).

Moving along, we get the maps size and set it to a variable B (3) and use it to calculate the measure, we can see that it is not affected by the extra pairs, just by the extra identifier in the *IOMs* (b): Measure = $((B / A = 3 / 5 = 0.6) * 0.18) = 0.108$.

Now, the algorithm will add the Candidates with a similar number of occurrences by category to a map:

Suspects {

- "main":["main"],

4.4. Methodology

}

Of course, since the source codes are quite different, most of the pairings were removed. From here, we calculate its number of identifiers ($C = 1$) and make another addition to the measure: $\text{Measure} = ((C/B = 1/3 = 0.333) * 0.42) = 0.14$.

The algorithm will now verify the node behaviors and add the similar ones to a new map:

Equivalences {

- "main": "main",

}

We can see that, both variables "x" and "y" are not equivalent between the source codes as they both have one more and one less occurrences, respectively. This results in a lower value for the number of identifiers in the Equivalences map ($D = 1$). That number is then used to calculate one more addition to the measure: $\text{Measure} = ((D/C = 1/1 = 1) * 0.38) = 0.38$.

Finally, the algorithm will compare the contents of the pairs in the Equivalences map and add the exact copies to another map:

Copies {

- "main": "main".

}

We now set a variable to this maps size ($E = 1$) and make a final addition of: $\text{Measure} += ((F/E = 1/1 = 1) * 0.02) = 0.02$,

This gives us a final measure of $0.108 + 0.14 + 0.38 + 0.02 = 0,648$ for this algorithm, which is a similarity measure indicating there are several differences between the ways identifiers were used in both *ASTs*.

4.4.4 *Scope changed*

Since Scope is not a part of our targets (see Section 3.1) we will also ignore it.

4.4.5 *Algorithm 3: Operands order switched*

Similar to the "Identifiers changed" type (see Section 4.4.3) we make a pair of maps and filter the information to compare them. However, this time we are looking for expressions and their respective parts.

1. Map each expression element to its occurrences (EOM_1, EOM_2),
2. Calculate the highest number of expressions between the EOMs (A),
3. For each pair of expression elements between the EOMs,

4.4. Methodology

- a) If their number of occurrences is similar,
 - i. Add the pair to a map (Candidates),
4. Calculate the number of expression elements in the Candidates map (B),
5. Measure += $(B / A) * 0.18$,
6. For each pair of expression elements in the Candidates map,
 - a) If their number of occurrences by category is similar,
 - i. Add the pair of expression nodes to a map (Suspects),
7. Calculate the number of expression elements in the Suspects map (C),
8. Measure += $(C / B) * 0.42$,
9. For each pair of expression elements in the Suspects map,
 - a) If they have a similar behavior,
 - i. Add the pair of to a map (Equivalences),
10. Calculate the number of expression elements in the Equivalences map (D),
11. Measure += $(D / C) * 0.38$,
12. For each pair of expression elements in the Equivalences map,
 - a) If their contents are the same,
 - i. Add the pair of to a map (Copies),
13. Calculate the number of expression elements in the Copies map (E),
14. Measure += $(E / D) * 0.02$,
15. Return Measure.

In this case, we map expression elements to their occurrence Node(s). This allows us to sort the expression elements and compare them regardless of their order. The similarity is verified by checking if the occurrence nodes have non-identifier neighbors with the same contents. To illustrate the algorithm, let us consider a comparison between source codes 3A and 3B (see Listings 4.7 and 4.8).

```
1 public class Example {
2     public void main() {
3         int x = 2;
4         int y = x+5;
5         return y;
6     }
7 }
```

Listing 4.7: Source code 3A.

```
1 public class Example {
2     public void main() {
3         int x = 2;
4         int y = 5+x;
5         return y;
6     }
7 }
```

Listing 4.8: Source code 3B.

4.4. Methodology

Looking at the *ASTs* that were generated from the previous source codes (in Figs. 13 and 14) we get a clear look at the input that the algorithm receives.

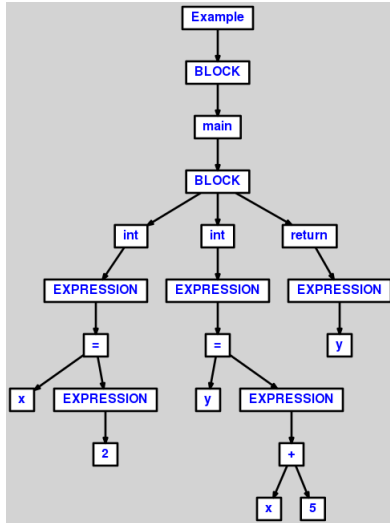


Figure 13.: *AST* generated from source code 3A.

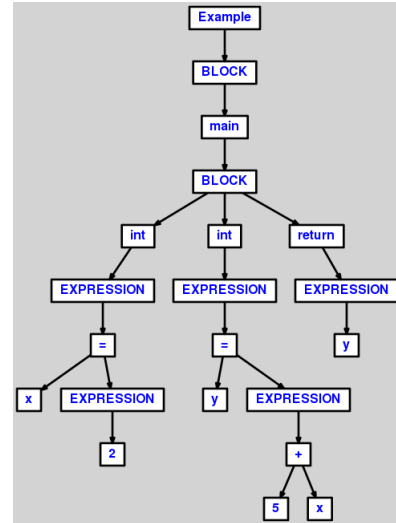


Figure 14.: *AST* generated from source code 3B.

We can see that the only change was from "x+5" to "5+x", meaning that the source codes have the same functionality.

The algorithm starts by creating a map between an expression element and its occurrences (EOM):

- EOM_{3A} : "=":[3,4], "y":[4,5], "x":[3,4], "2":[3], "+":[4], "5":[4],
- EOM_{3B} : "=":[3,4], "y":[4,5], "x":[3,4], "2":[3], "+":[4], "5":[4].

Note that expressions can have: logical operators (such as *AND*), relational operators (such as *>*) or primitives (identifiers or values).

In this case, it is clear that both *ASTs* have nodes with the same contents with 2 of them switched (namely: a "x" and "5" node).

The next step is calculating the size of the bigger EOM which, in this case, is the same for both of them ($A = 6$).

The algorithm will calculate the number of occurrences for each pair of expressions between the EOMs and add the similar ones to a map. The contents of the Candidates map would be as follows:

Candidates {

- ("=", "x", "y"):["=", "x", "y"],
- ("2", "+", "5"):["2", "+", "5"]

}

The number of expressions (on the left side) is then calculated ($B = 6$) and the measure is set to an initial value of $(B/A = 6/6 = 1) * 0.18 = 0.18$.

4.4. Methodology

At this point, each pair in the list of Candidates is compared by the number of occurrences by type and those that match are added to a new map:

Suspects {

- "=":["="],
- "x":["x"],
- "2":["2"],
- "y":["y"],
- "+"":["+"],
- "5":["5"]

}

The number of copies is later calculated ($C = 6$) and the measure is increased by $(C / B = 6 / 6 = 1) * 0.42 = 0.42$.

Now each pair of nodes is checked for similarity by comparing some of their contents and the structure beneath them. These nodes are added to a new map:

Equivalences {

- "=":["="],
- "x":["x"],
- "2":["2"],
- "y":["y"],
- "+"":["+"],
- "5":["5"]

}

As we can see, this map still has the "+" nodes associated. This is because operations of addition (+) or multiplication (*) have the commutative property, meaning that swapping their values will not change the result.

The number of equivalences is later calculated ($D = 6$) and the measure is increased by $(D / C = 6 / 6 = 1) * 0.38 = 0.38$.

Finally, the names and values of each pair are checked to see if they are exact copies and the matching pairs are added to a map:

Copies {

- "=":["="],

4.4. Methodology

- "x":["x"],
- "2":["2"],
- "y":["y"],
- "+"":["+"],
- "5":["5"]

}

As we can see, the "=" and "+" nodes are not exact copies. This was due to the fact that their children are also checked and must maintain their order.

From here, the number of equations in the map of Copies is calculated ($E = 6$) and it is used to make a final addition of $(E/D = 6/6 = 1) * 0.02 = 0.02$ to the measure.

Finally, the similarity measure calculated by this algorithm is produced which resulted in a value of $0.18 + 0.42 + 0.38 + 0.02 = 1.0$. This value clearly shows that there are hardly any differences between the source codes in terms of expressions.

This algorithm used several metrics, with each of them being stricter which resulted in an accurate final measure indicating that they are similar but not the same.

To try something harder, let's compare the previous source code 3A to a new source code 3C (see Listings 4.9 and 4.10).

```
1 public class Example {
2     public void main() {
3         int x = 2;
4         int y = x+5;
5         return y;
6     }
7 }
```

Listing 4.9: Source code 3A.

```
1 public class Copy {
2     public void main() {
3         int a = 2;
4         int c = 5;
5         int b = 5+a;
6         return b;
7     }
8 }
```

Listing 4.10: Source code 3C.

Once again, we will also show the *ASTs* that were generated from source codes 3A and 3C (see Figs. 15 and 16).

4.4. Methodology

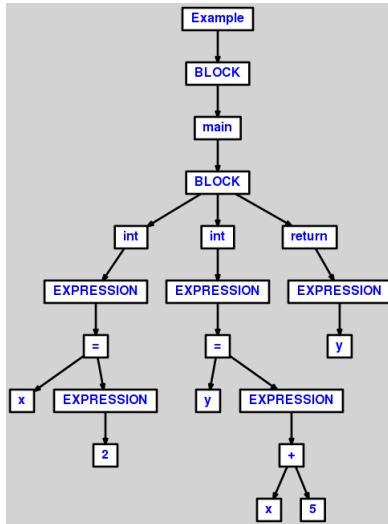


Figure 15.: AST generated from source code 3A.

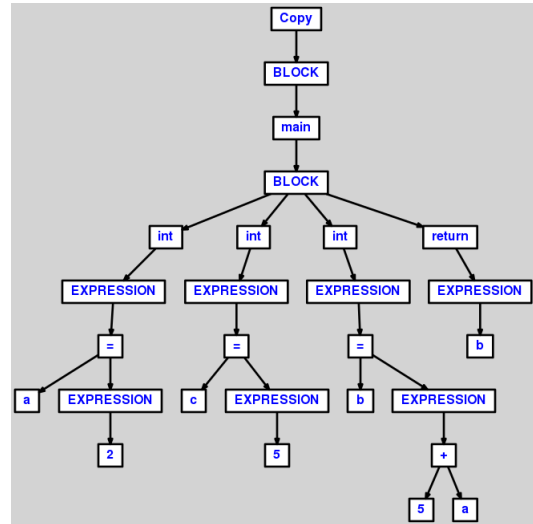


Figure 16.: AST generated from source code 3C.

As we can see, a "c=5" was added and the identifiers were changed but the functionality is the same. The algorithm would, once again, start by creating a map between each expression element and its occurrences (EOM):

- $EOM_{3A}: \{ "=" : [3,4], "y" : [4,5], "x" : [3,4], "2" : [3], "+" : [4], "5" : [4] \}$
- $EOM_{3C}: \{ "=" : [3,4,5], "b" : [5,6], "a" : [3,5], "2" : [3], "c" : [4], "5" : [4,5], "+" : [5] \}$

As expected, the higher number of expressions between the EOMs increased to (A =) 7 due to the extra "c=5".

This adds more pairs when verifying and adding the nodes with the same number of occurrences to a map:

Candidates {

- ("2", "5", "+"): ["2", "c", "+"],
- ("y", "=", "x"): ["b", "5", "a"]

}

Since some nodes now have additional occurrences, we can see that pairs such as "y": "c" are not similar enough, which would result in less matches and a lower measure overall. If we do not want the comparison to be so strict we could set the threshold to 0,67 (2/3) which would have changed the map into the following:

Candidates {

- ("2", "5", "+"): ["2", "c", "+"],
- ("y", "=", "x"): ["2", "b", "c", "5", "a", "+", "="]

4.4. Methodology

}

As we can see, the number of elements is still 6 but they are associated to far more nodes.

Let us carry on with the first map of Candidates. We set the number of expression elements (6) to variable B. Note that if we were to compare source code 3C with 3A instead, the value of B would increase, which makes results asymmetrical.

The measure would then be set to an initial value of $(B / A = 6 / 7 = 0.857) * 0.18 = 0.154$.

At this point, each pair in the list of Candidates is compared by the number of occurrences by category and those that match are added to a new map:

Suspects {

- "y":["b"] (Identifier, Under "=" and "return"),
- "x":["a"] (Identifier, Under "=" and "+"),
- "2":["2"] (Number, Under "="),
- "+"":["+"] (Operation, Under "=")

}

Note that, since this comparison is "one to many" we also check the parents types to make sure that the expressions are under the same context.

The number of Suspects (C = 4) is then used to calculate an increase of the measure of $(C / B = 4 / 6 = 0.667) * 0.42 = 0.28$.

Once again the next step is comparing the similarity between each pair of Suspect nodes by checking some of their contents and behavior. Which leads to a new map:

Equivalences {

- "y":["b"],
- "x":["a"],
- "2":["2"],
- "+"":["+"]

}

Note that, the information from the previous algorithm would be used along with the comparison of some contents like primitives or operations to reach this result.

Looking at the list of Equivalences, we can use its size (D = 4) to make yet another addition to the measure: $(D / C = 4 / 4 = 1) * 0.38 = 0.38$.

Finally, we check if the nodes contents are exact copies. In which case we add them to a map (Copies):

- "2":["2"],

4.4. Methodology

- "+":["+"]

Which has a total of 2 expressions elements ($E = 2$) and gives an addition of $(E/D = 2/4 = 0.5) * 0.02 = 0.01$ to the measure.

We can see that it uses the same strategy: filter the data several times to get accurate results. This gives us a final measure of $0.1548 + 0.28 + 0.38 + 0.01 = 0.825$.

4.4.6 Algorithm 4: Variable types and control structures replaced

In this case we can ignore variable types as they are not our targets (see Section 3.1) but must still check control structures.

1. Map control structures to their conditions (CCM_1, CCM_2),
2. Calculate the highest number of conditionals between the CCMs (A),
3. For each pair of conditional nodes between the CCMs,
 - a) If the number of nodes in their conditions is similar,
 - i. Add the pair to a map (Candidates),
4. Calculate the size of the Candidates map (B),
5. Measure += $(B/A) * 0.18$,
6. For each pair of conditional nodes in the Candidates map,
 - a) If they have the same number of nodes by category,
 - i. Add the pair of expression nodes to a map (Suspects),
7. Calculate the size of the Suspects map (C),
8. Measure += $(C/B) * 0.42$,
9. For each pair of conditional nodes in the Suspects map,
 - a) If they have a similar behavior,
 - i. Add the pair of to a map (Equivalences),
10. Calculate the size of the Equivalences map (D),
11. Measure += $(D/C) * 0.38$,
12. For each pair of expression nodes in the Equivalences map,
 - a) If their contents are the same,

4.4. Methodology

- i. Add the pair of to a map (Copies),
13. Calculate the size of the Copies map (E),
14. Measure += (E / D) * 0.02,
15. Return Measure.

In this case, Conditional nodes are considered similar if their children nodes have (non-identifier) nodes with the same contents.

We want to map each Conditional (control structure) node to its conditions nodes.

Let us start by defining two source codes with an equivalent functionality (see Listings 4.11 and 4.12).

```

1 public class Example {
2   public void main() {
3     int a = 5;
4     if(a == 5) {
5       return a;
6     }
7     return 0;
8   }
9 }

```

Listing 4.11: Source code 4A.

```

1 public class Example {
2   public void main() {
3     int a = 5;
4     while(5 == a) {
5       return a;
6     }
7     return 0;
8   }
9 }

```

Listing 4.12: Source code 4B.

As usual, ASTs are generated from the source codes, which gives us an abstract perspective (see Figs. 17 and 18).

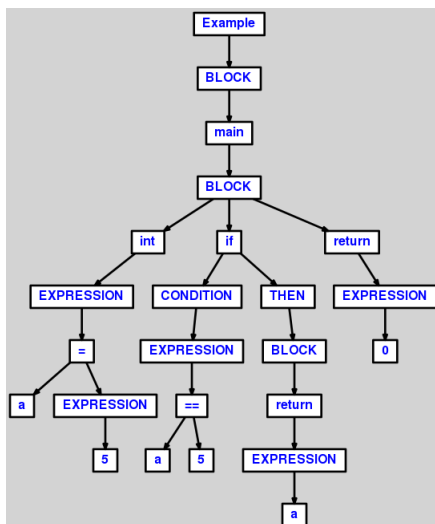


Figure 17.: AST generated from source code 4A.

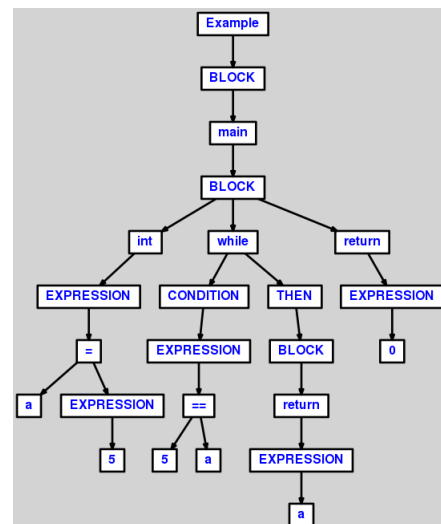


Figure 18.: AST generated from source code 4B.

We can see that this time, the "if" conditional was replaced with a "while" but the functionality is the same, this is due to their block only having a "return" node, which stops the loop.

4.4. Methodology

If we now use this algorithm to compare source codes 4A to 4B, we start by mapping the control structure nodes to their respective conditions (CCM):

- CCM_{4A} : {if:[a, ==, 5]},
- CCM_{4B} : {while:[5, ==, a]}

We can see that, unlike the previous maps, these maps are small ($A = 1$) since there is only 1 conditional in each source code.

Looking their conditions, we can see that they are exact copies, meaning that we add them to a map of Candidates:

- if:[while]

As expected, it is a map with a single element ($B = 1$). This gives the measure an initial increment of $(B/A = 1/1 = 1) * 0.18 = 0.18$.

These conditions are then compared in terms of the number of nodes per type which is similar, thus being added to a map of Suspects:

- if:[while]

Once again, as these conditions are exact copies, they have the same number of nodes by type ($C = 1$). This results in an increase of $(C/B = 1/1 = 1) * 0.42 = 0.42$ to the measure.

At this point, the nodes are checked for similarity, meaning that some of their contents are compared as well as the functionality. As this switch (if to while) can break the functionality, it is important to note that the block that follows must be checked for calls to "return" and/or "break". This is what allows the "while" loop to run a single time, which is what an "if" conditional does.

The map of Equivalences will therefore be the same as the previous maps:

- if:[while]

Using the previous maps size ($D = 1$), another addition is made to the measure: $(D/C = 1/1 = 1) * 0.38 = 0.38$.

Which leaves us with one final target: exact copies. For each pair in the Equivalences map, the contents of the conditionals and the conditions are compared, leaving us with an empty map of exact copies. This means that the size of the Copies map ($E = 0$) is zero, as well as the last increment to the measure: $(E/D = 0/1 = 0) * 0.02 = 0$.

This results in a measure of $0.18 + 0.42 + 0.38 + 0 = 0.98$. This result was to be expected considering how close the source codes were but we must note that unless the sources are exact copies, the measure will never reach 100%.

4.4. Methodology

To make things interesting, let us consider a comparison between "if" conditionals and "case" conditionals, as seen in Listings 4.13 and 4.14.

```

1 public class Example {
2   public void main() {
3     int a = 5;
4     if(a == 5) {
5       System.out.println("Hello");
6     }
7     if(a == 7) {
8       System.out.println("World");
9     }
10  }
11 }

```

Listing 4.13: Source code 4C.

```

1 public class Example {
2   public void main() {
3     int a = 5;
4     switch(a) {
5       case 5:
6         System.out.println("Hello");
7         break;
8       case 7:
9         System.out.println("World");
10        break;
11    }
12  }
13 }

```

Listing 4.14: Source code 4D.

Once again, an AST is generated from each source codes (see Figs. 19 and 20).

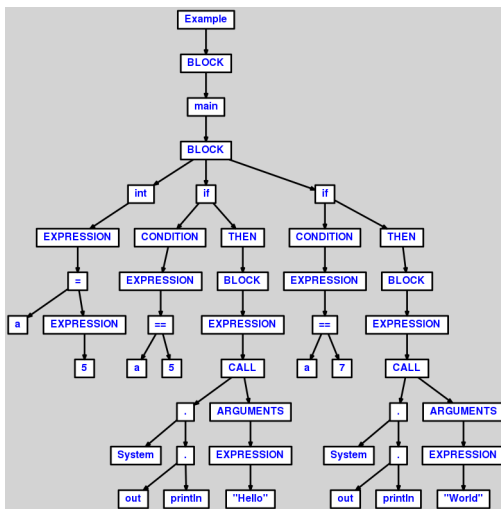


Figure 19.: AST generated from source code 4C.

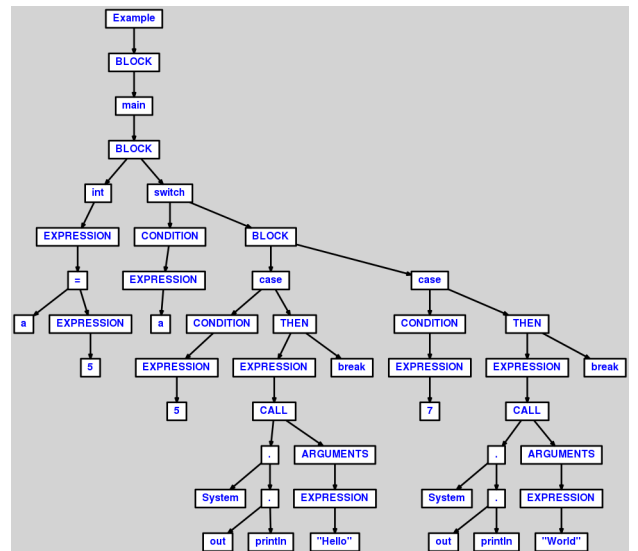


Figure 20.: AST generated from source code 4D.

As stated, we are comparing a source code with two "if" conditionals to a "switch" node with two "case" conditionals. Note that the condition associated to a "case" conditional equates to: switch condition == case condition. Meaning that they are a special type of conditional.

Using the algorithm to compare source code 4C to 4D, we would once again start by building the CCMs:

- CCM_{4C} : {if:[a, ==, 5], if:[a, ==, 7]}
- CCM_{4D} : {case:[5], case:[7]}

4.4. Methodology

We can see that the "case" conditionals were only associated to 5 and 7 instead of the "if" conditionals. This is because, whenever we compare things to Case conditionals we build an artificial condition which has a "==" node and the switch and case conditions as its children.

The highest number of nodes in the CCMs is now evaluated ($A = 2$) and, each pair of conditionals between them is checked. If their conditions have an equal number of nodes, then we add them to a new map:

Candidates {

- if:[case], if:[case],
- if:[case], if:[case]

}

In this case, there is a need for the removal of any break nodes since they are mandatory for a "case" to be equivalent to an "if". This results in this map having all of the previous nodes ($B = 2$), which is used in the calculations to increase the measure: $(B / A = 2 / 2 = 1) * 0.18 = 0.18$.

We now compare the number of nodes (in each condition) per type and add the similarities to a new map:

Suspects {

- if:[case], if:[case],
- if:[case], if:[case]

}

Once again, the conditions in the "if"s are compared to artificial conditions which have the "switch" and "case" conditions, along with a parent "==" node. This allows for the comparison to recognize that these conditionals are indeed equivalent. Resulting in a number of nodes by type of ($C =$) 2. Which leads to an increase of $(C / B = 2 / 2 = 1) * 0.42 = 0.42$ in the measure.

We now check the nodes for similarity, this includes comparing the contents of some nodes and checking the previous maps when necessary. Of course, this time we can see that this is the point where the map will be smaller since we know that two of the conditionals have a "5" node and the other two have a "7". The resulting map gives us a good estimate of the similar nodes:

Equivalences {

- if:[case], if:[case],
- if:[case], if:[case]

}

We are left with a map that accurately associates conditionals in both source codes. The number of comparisons is then calculated ($D = 2$) and the measure is increased once again: $(D / C = 2 / 2 = 1) * 0.38 = 0.38$.

4.4. Methodology

At this point, we already detected that the "if" and "case" nodes are equivalent but they are not equal. Either way, the algorithm has this final step for the cases where they are indeed exact copies by checking if the conditionals and their conditions have the same contents and adding those that do to a map (Copies).

This map will be empty ($E = 0$) and the final increment will be if $(E/D = 0/2 = 0) * 0.02 = 0$, which results in a final measure of: $0.18 + 0.42 + 0.38 + 0 = 0.98$.

4.4.7 *Statements order switched*

As Statements are not a part of our targets (see Section 3.1) we will ignore them and let the next method handle such cases.

4.4.8 *Algorithm 5: Group of calls turned into a function call or vice versa*

This algorithm was made to detect the similarities between functions and blocks of source code (usually between a "{" and a "}"). To do so, the algorithm has to analyze all the child nodes within the blocks.

1. Map blocks to their header name⁴ (BNM_1, BNM_2),
2. Map block nodes to their children⁵ (BCM_1, BCM_2),
3. For each block node in the BCM,
 - a) If the number of children (from both BCMs) is similar,
 - i. Add the block node to a map (Candidates),
4. Calculate the size of the Candidates map (B),
5. Measure += $(B / A) * 0.18$,
6. For each pair in the Candidates map,
 - a) If the number of nodes by category inside the blocks is similar,
 - i. Add the pair to a map (Suspects),
7. Calculate the size of the Suspects map (C),
8. Measure += $(C / B) * 0.42$,
9. For each call node in the Suspects map,

4 This is the name of the blocks parent, in other words, the name of the class/method which owns this block (for ex.: The block in "main..." has "main" as its header name).

5 The children are: every node inside the block along with the nodes from called blocks.

4.4. Methodology

- a) If the associated blocks have a similar behavior,
 - i. Add the pair to a map (Equivalences),
10. Calculate the size of the Equivalences map (D),
11. Measure += (D / C) * 0.38,
12. For each call node in the Equivalences map,
 - a) If the calls are exact copies,
 - i. Add the pair to a map (Copies),
13. Calculate the size of the Copies map (E),
14. Measure += (E / D) * 0.02,
15. Return Measure.

Note that the BNMs are only used to build the BCMs as any call node to a local method that they have will have its block added to the children.

As a test case, let us consider the source codes in Listings 4.15 and 4.16.

```
1 public class Hello {
2     public void main() {
3         int b = 7;
4         int a = 5;
5         System.out.println("Hello World");
6     }
7 }
```

Listing 4.15: Source code 5A.

```
1 public class Hello {
2     public void hello() {
3         int a = 5;
4         System.out.println("Hello World");
5     }
6     public void main() {
7         int b = 7;
8         hello();
9     }
10 }
11 }
```

Listing 4.16: Source code 5B.

Figure 21.: Source codes 5A and 5B, respectively.

The ASTs that were generated from them can be seen in Figs. 22 and 23.

4.4. Methodology

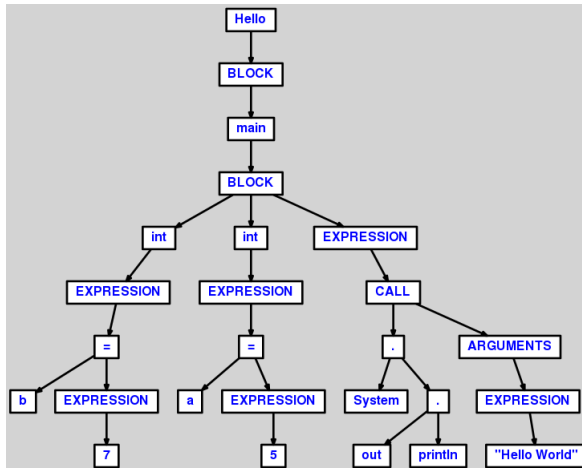


Figure 22.: AST generated from source code 5A.

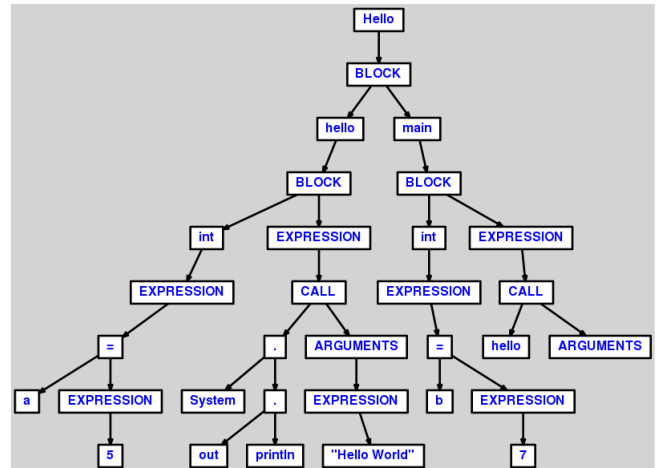


Figure 23.: AST generated from source code 5B.

This time, we start by making a map of blocks and their names (BNM) and then a map of blocks and their children (BCM).

- BNM_{5A} : {Hello:"Hello", main:"main"}
- BNM_{5B} : {Hello:"Hello", main:"main", hello:"hello"}
- BCM_{5A} : {
 - Hello:[main, int, =, b, 7, int, =, a, 5, System.out.println_{call}, "Hello World"],
 - main:[int, =, b, 7, int, =, a, 5, System.out.println_{call}, "Hello World"]
- BCM_{5B} : {
 - Hello:[hello, int, =, a, 5, System.out.println_{call}, "Hello World", main, int, =, b, 7, int, =, a, 5, CALL (System.out.println), "Hello World"],
 - main:[int, =, b, 7, int, =, a, 5, System.out.println_{call}, "Hello World"]
 - hello:[int, =, a, 5, System.out.println_{call}, "Hello World"]

Note that the "Hello" blocks (representing Classes) have every single node except for themselves as children. Also, the "main" block in BNM_{5B} calls "hello" but the call node was replaced with the children of its block. Of course, if it is a call to an external method like System.out.println we simply leave the call node.

We then calculate the total number of blocks in both BCMs ($A = 3$).

Now, we compare the number of children in each pair of blocks and add the similarities to a map:

Candidates {

4.4. Methodology

- main:[main]

}

We then calculate the size of the Candidates map ($B = 1$) and make an addition of $(B / A = 1 / 3 = 0.667) * 0.18 = 0.12$ to the measure.

The algorithm then proceeds as usual, by comparing the number of nodes by category and adding the similar pairs to a map:

Suspects {

- main:[main]

}

Then calculating the size of the map ($C = 1$) and making another addition to the measure: $(C / B = 1 / 1 = 1) * 0.42 = 0.42$

Proceeding with the comparison of the nodes similarity and adding the similar ones to a map:

Equivalences {

- main:[main]

}

And making another addition to the measure using the maps size ($D = 1$): $(D / C = 1 / 1 = 1) * 0.38 = 0.38$

Finally, we check each pair of blocks has the same name and add it to a map:

Copies {

- main:[main]

}

$(D / C = 1 / 1 = 1) * 0.02 = 0.02$

Finally, we calculate the full measure and get $0.18 + 0.42 + 0.38 + 0.02 = 0.88$. Which means that some of the blocks in these source codes are equal.

Now, let us consider a case where the calculations inside a return are replaced with a function and its arguments are used to pass the parameters (see Listings 4.17 and 4.18).

```
1 public class Addition {
2     public void main() {
3         return 5 + 10;
4     }
5 }
```

Listing 4.17: Source code 5C.

```
1 public class Addition {
2     public int sum(int a, int b) {
3         return a + b;
4     }
5
6     public void main() {
7         return sum(5, 10);
8     }
9 }
```

Listing 4.18: Source code 5D.

4.4. Methodology

Observing the source codes it does not look like a big change but, when we look at the ASTs that are generated (see Figs. 24 and 25) from those source codes, we can see how much changed in the structure.

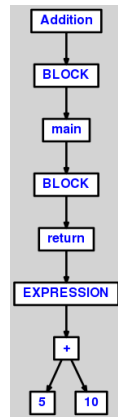


Figure 24.: AST generated from source code 5C.

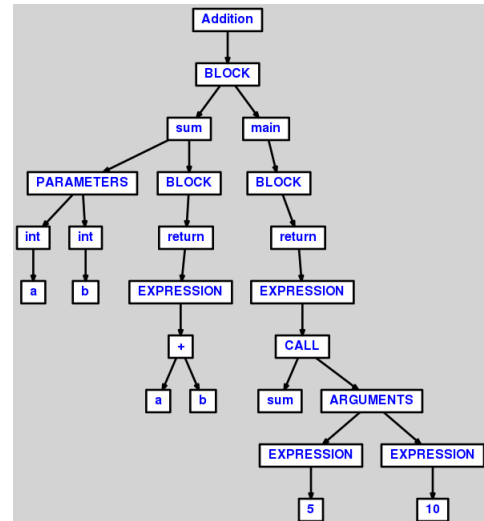


Figure 25.: AST generated from source code 5D.

The algorithm starts by building maps that associate blocks to names (BNM) and uses them to build the maps that associate blocks to their children (BCM):

5: Have Suspicious Blocks ? Yes (88)

- BNM_{5A} : {Addition:"Addition", main:"main"}
- BNM_{5B} : {Addition:"Addition", main:"main", sum:"sum"}
- BNM_{5A} : {
 - Addition:[main, return, +, 5, 10],
 - main:[return, +, 5, 10]
- BNM_{5B} : {
 - Addition:[sum, int, a, int, b, int, return, +, a, b, main, return, return, +, a, b],
 - main:[return, return, +, a, b],
 - sum:[return, +, a, b]

In this case, we can see that source code 5C has no calls and that 5D calls the sum method once.

4.4. Methodology

We then calculate the higher number of blocks in the BCMs and set it to variable A (= 1). Then add each pair of blocks that has the same number of children to a map:

Candidates {

- main:sum,
- Addition:main

}

Note that the map is missing the pairs that are actually similar. This is, once again, due to the comparisons strictness and could be changed by setting a threshold.

We set this maps size (2) to a variable B and make an addition to the measure of $(B / A = 2 / 3 = 0.667) * 0.18 = 0.12$

We then compare the number of nodes by category and add the blocks with similarities to a map. Of course, looking at the BCMs we can tell that none of the pairs have the same number of nodes by category so we do not get any additions as all of the following maps is empty.

We end with a total measure of $0.12 + 0 + 0 + 0 = 0.12$ which shows that these blocks are quite different.

4.4.9 Main Algorithm

As stated at the beginning of this Section, each method produces a measure which is used to calculate a final measure. This measure is an overall similarity measure that compares a pair of Suspects.

Let us consider that each algorithm was implemented in a method Method_{*i*}, where *i* is a number from 1 to 5. Let us also consider that this algorithm takes 2 Suspects that will be represented as S_a and S_b.

The main algorithm would work as follows:

1. M = Array with 5 elements,
2. M[1] = Method₁(S_a,S_b),
3. M[2] = Method₂(S_a,S_b),
4. M[3] = Method₃(S_a,S_b),
5. M[4] = Method₄(S_a,S_b),
6. M[5] = Method₅(S_a,S_b),
7. If M[1] is different from 0,
 - a) Return M[1].
8. Otherwise

4.4. Methodology

- a) Calculate the number of Ms that are not 0 (A),
- b) Measure = $\left(\frac{M[2] + M[3] + M[4] + M[5]}{A}\right) * 100$,
- c) Return Measure.

As we can see, the algorithm either returns the M[1] measure (which is either 0% or 100%) or the computation done using the other results (M[2] to M[5]) that were not 0%. We only consider the measure if it was able to calculate a result.

SPECTOR TESTS

The resulting Spector tool is housed in <http://www.di.uminho.pt/~gepl/Spector/> and is open source in an effort to make an *AST* tool available.

It supports the following languages:

- Java

Like the previous tools, we tested it against the Calculator Java source codes (see Appendix B.8.1) and added the resulting measures to the overview which you can see in Fig. 26.

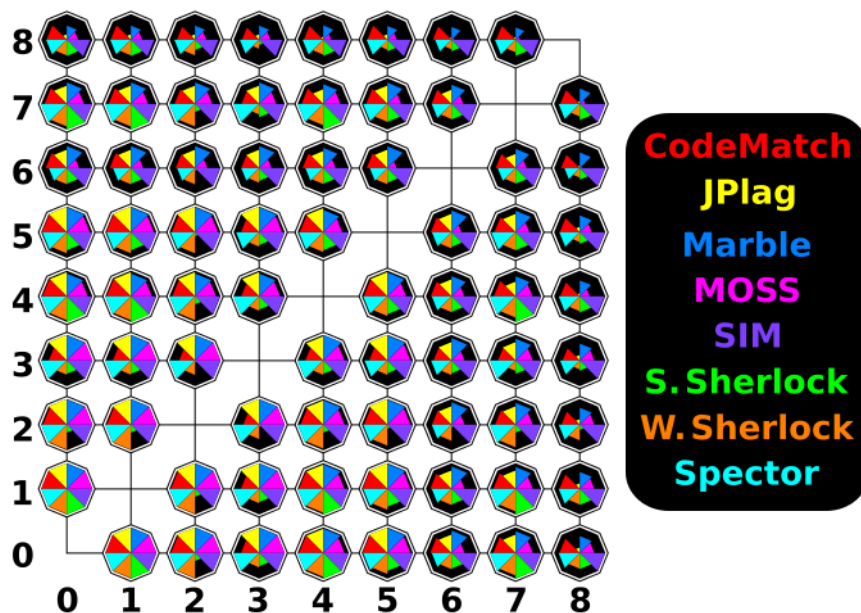


Figure 26.: An overview of the results obtained for the Calculator, Java source codes.

The results are good for all of the smaller changes (from 1 to 4) and the remaining results are close to the ones returned by Code Match.

To test how long the operations take, we also used the "-u sdhai" arguments, which makes Spector produce every output possible as well as time (the *i* output) each one of them. The following were the times reported:

- Listing the input files took 0.039s
- Parsing and transforming input into ASTs took 1.306s
- Building visual ASTs took 30.179s
- Comparing suspects while printing details took 3.767s
- Producing HTML results took 0.048s
- Printing the summary took 0.036s

As we can see, Spector took less than 6 seconds (5.148s) to list, parse+transform, compare the suspects and print the details and summary. If we consider the fact that it compared 36 pairs of source codes, and that its code is not optimized, we could say that it is reasonably fast. Of course, the whole operation took around half a minute (35.375s) but the time sink was the creation of the visual ASTs. Among the results, an HTML (the *h* output) file was produced which presents the summary (see Fig. 27). Its a simple but effective presentation and much like the textual results, it shows the results sorted in an order. Note that the order was set to ascending, to show some variety in the results.

Spector

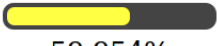
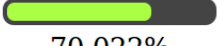
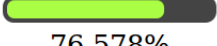
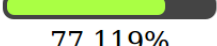
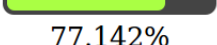
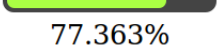
First File	Second File	Similarity
L6.java	L8.java	 59.954%
L6.java	L7.java	 70.022%
L5.java	L6.java	 76.578%
L3.java	L8.java	 77.119%
L3.java	L6.java	 77.142%
L0.java	L6.java	 77.363%

Figure 27.: Part of the HTML presenting the results for the Calculator, Java source codes.

If we inspect the more complex 21 Matches, Java source codes using the same options we get larger delays:

- Listing the input files took 0.011s

- Parsing and transforming input into ASTs took 1.521s
- Building visual ASTs took 53.036s
- Comparing suspects while printing details took 14.251s
- Producing HTML results took 0.009s
- Printing the summary took 0.038s

But, once again, we can see that the visual ASTs took the longest to produce. This time, the operation took around 1 minute (1m8.866s) but if we take away the AST and HTML results, we get a runtime of 16 seconds (15.821s) which is expectable considering that these source codes are about 2.5 times bigger than the Calculator ones.

Looking at Fig. 28, we can see the resulting HTML from the previous test.

Spector




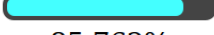
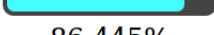
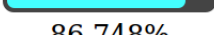
First File	Second File	Similarity
L5.java	L8.java	 85.339%
L3.java	L5.java	 85.649%
L1.java	L5.java	 85.762%
L2.java	L5.java	 85.762%
L0.java	L5.java	 86.445%
L5.java	L6.java	 86.748%

Figure 28.: Part of the HTML presenting the results for the 21 Matches, Java source codes.

All of these results can be found at the dissertations website: <http://www.di.uminho.pt/~gepl/Spector/paper/>.

CONCLUSIONS AND FUTURE WORK

In this dissertation, plagiarism in software was introduced and characterized as a problem. We observed the existing tools and saw that they usually have problems with code where plagiarism was dissimulated in certain ways. This proved that there is a need for a new tool and gave us the motivation to build a tool capable of detecting plagiarism. We chose to use a structure based methodology with *ASTs* as the abstract structure and found some papers about such tools that showed promising results. Our purpose is not to build a tool that will detect plagiarism in every case as that is implausible, but instead to build a tool that will help teachers find cases of plagiarism and dissuade students from plagiarizing. However, as the algorithms were aimed at detecting similarities, the tool inevitably produces false positives in cases like when a group of students does the same exercise, for instance. Looking back at the algorithms (see Section 4.4), we can see that this tool is more of a source code **similarity** detector. It does not avoid false positives, it simply has a total measure for each step/level of similarity (candidates, suspects, equivalences, copies). Our contribution is therefore not a plagiarism detector which, in our opinion, depends on the teachers/schools interpretation of what plagiarism is but a structure-based similarity detector that is open source and can be extended to execute different tests, tailored to a desired paradigm.

Future work could involve the following improvements for the tool:

- Detection of additional cases (for example: do-while conditionals)
- Showing the matched pairs directly on the source codes
- Excluding base code from the inspection
- Production of HTMLs presenting detailed results
- Options for modifying the weights used in the comparisons
- Support for other programming languages
- A GUI to facilitate interaction with the tool
- Testing different cases to optimize the threshold and weights

Naturally, as *ANTLR* evolves, the tools can be updated to use newer versions and the updated grammars (for newer versions and additional languages) associated. Note however that tree reconstruction¹ in *ANTLR4* requires more implementations to work.

Looking at the architecture (in Fig. 3), we can see that this methodology can also be used for different detections, such as:

- Detecting plagiarism while trying to avoid false positives
- Detecting copies within the same source codes (similar to CPD Copeland (2003))
- Detecting similarity between source codes using different programming languages

In conclusion, the results have shown that using a structure-based AST comparison is feasible in terms of accuracy and time and that, given further tests and adjustments, the tool will do a good job when we need to check how similar a pair of Java source codes are.

¹ As seen in <http://stackoverflow.com/questions/14565794/antlr-4-tree-inject-rewrite-operator>.

BIBLIOGRAPHY

- Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, Baltic Sea '06, pages 141–142, New York, NY, USA, 2006. ACM. doi: 10.1145/1315803.1315831. URL <http://doi.acm.org/10.1145/1315803.1315831>.
- I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998. doi: 10.1109/ICSM.1998.738528.
- Andrés M. Bejarano, Lucy E. García, and Eduardo E. Zurek. Detection of source code similitude in academic environments. *Computer Applications in Engineering Education*, 2013. ISSN 1099-0542. doi: 10.1002/cae.21571. URL <http://dx.doi.org/10.1002/cae.21571>.
- Tom Copeland. Detecting duplicate code with PMD's CPD, 2003. URL http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.html.
- G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Trans. on Educ.*, 51(2): 195–200, May 2008. ISSN 0018-9359. doi: 10.1109/TE.2007.906776. URL <http://dx.doi.org/10.1109/TE.2007.906776>.
- Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, pages 668–673, 2010. doi: 10.1109/ICBNMT.2010.5705174.
- J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11:11–19, 1987. doi: 10.1016/0360-1315(87)90042-X.
- Daniela Fonte, Ismael Vilas Boas, Daniela da Cruz, Alda Lopes Gançarski, and Pedro Rangel Henriques. Program analysis and evaluation using quimera. In José Cordeiro, Lesvek Maciazek, and Alfredo Cuzzocrea, editors, *ICEIS'2012 — 14th International Conference on Enterprise Information Systems*, pages 209–219. INSTICC – Institute for Systems and Technologies of Information, Control and Communication, June 2012. doi: 10.4320/OASICS.SLATE.2012.I. also available from SciTePress Digital Library.
- Dick Grune and Matty Huntjens. Het detecteren van kopieën bij informatica-practica. *Informatie*, 31 (11):864–867, 1989.

Bibliography

- Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28, 2010.
- Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4): 264–268, 1978.
- Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE TRANSACTIONS ON EDUCATION*, 42(2):129–133, May 1999. URL <http://wrap.warwick.ac.uk/14558/>.
- Xiao Li and Xiao Jing Zhong. The source code plagiarism detection using AST. In *Intelligence Information Processing and Trusted Computing (IPTC), 2010 International Symposium on*, pages 406–408, 2010. doi: 10.1109/IPTC.2010.90.
- Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pages 872–881. ACM Press, 2006.
- A. Parker and J. O. Hamblen. Computer algorithms for plagiarism detection. *Education, IEEE Transactions on*, 32(2):94–99, 1989. ISSN 0018-9359. doi: 10.1109/13.28038.
- Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, 1995. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.70>.
- Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultät für Informatik, Universität Karlsruhe, 2000.
- Saul Schleimer. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003.
- Ilana Shay, Nikolaus Baer, and Robert Zeidman. Measuring whitespace patterns as an indication of plagiarism. In *Proceedings of the ADFSL Conference on Digital Forensics, Security and Law*, pages 63–72, 2010.
- Geoff Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138, 1990. ISSN 0164-1212. doi: [http://dx.doi.org/10.1016/0164-1212\(90\)90118-6](http://dx.doi.org/10.1016/0164-1212(90)90118-6). URL <http://www.sciencedirect.com/science/article/pii/0164121290901186>. jce:title;Special Issue on Using Software Metrics;/ce:title;.
- Michael J Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *ACM SIGCSE Bulletin*, volume 24, pages 268–271. ACM, 1992.

Bibliography

Michael J. Wise. *Running Karp-Rabin matching and greedy string tiling*. Basser Dept. of Computer Science, University of Sydney, Sydney, 1993. ISBN 0-86758-669-9.

Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, pages 130–134. ACM Press, 1996.



SOURCE CODES USED IN THE TESTS

This chapter presents the source files used in the tests. This includes both the tests built from an original source file (L0) and its variants (from L1 to L8) where each one corresponds to a different type of plagiarism (see section 2.3.4). As all the files are based on the same original, from a theoretical view every test would be a 100% match.

Source codes were made in both the Java and the C languages and are available online but only the Calculator, Java and the 21 Matches, C source codes will be presented in this document.

A.1 HIGHLIGHTS

The following highlights were used to show the changes that were done:

RED Text that was removed

YELLOW Text that was added or changed

BLUE, **GREEN** Text that was moved between the **blue** position and the following **green** position.

CYAN AND MAGENTA Block (for ex.: entire function) that was moved between the **cyan** position and the following **magenta** position.

A.2 THE PROGRAM, CALCULATOR

The goal in this program is to implement a simple calculator with the basic operations (addition, subtraction, division and multiplication) and a textual interface.

Only the Java version of the file was used to produce the results presented.

The following metrics characterise the original file:

- **Number of lines:** 82
- **Number of functions:** 1

A.2. The program, Calculator

- **Number of variables:** 6

The Java source codes files displayed here are available online at https://tinyurl.com/dopisiae/source/calculator_java.zip, as well as the C source code files, at https://tinyurl.com/dopisiae/source/calculator_c.zip.

A.2.1 Original source code for the Calculator exercise and the 1st type of plagiarism.

The 1st type of plagiarism (see Section 2.3.4) is an exact copy of the original.

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     public static void main(String args[])
6     {
7         // Operation (on) and Operands (od)
8         char on = ' ';
9         double od1 = 0.0, od2 = 0.0;
10
11         // Prepare a scanner to read the input
12         Scanner input = new Scanner(System.in);
13
14         // Show the program title
15         System.out.println("Calculator");
16
17         // Repeat until the user chooses to exit (x)
18         do {
19             on = ' ';
20             od1 = 0.0;
21             od2 = 0.0;
22
23             // Show the Menu
24             System.out.println();
25             System.out.println("+ Add");
26             System.out.println("- Subtract");
27             System.out.println("* Multiply");
28             System.out.println("/ Divide");
29             System.out.println("x Exit");
30             System.out.println();
31
32             // Ask for the users choice
33             System.out.print("Choice:\t\t");
34
35             // Read the input (a single character)
36             String temp;
37             try {
38                 temp = input.nextLine();
39                 on = temp.charAt(0);
40             } catch (Exception x) {
```


A.2. The program, Calculator

```
41     on = ' ';
42 }
43
44 if(on == '+' || on == '-' || on == '/' || on == '*') {
45     // Ask for and read the operands if the user chose an operation
46     System.out.print("Operand 1:\t");
47     od1 = input.nextDouble();
48     System.out.print("Operand 2:\t");
49     od2 = input.nextDouble();
50     input.nextLine();
51 } else if(on == 'x' || on == 'X') {
52     // Do nothing if the user wants to exit
53 } else {
54     // Give an error if its not a valid operation
55     System.out.println("Unknown Operation!");
56 }
57
58 // Execute operation and report result
59 switch(on) {
60     case '+': // Add
61         System.out.println("Result:\t\t"+(od1+od2));
62         break;
63
64     case '-': // Subtract
65         System.out.println("Result:\t\t"+(od1-od2));
66         break;
67
68     case '/': // Divide
69         if(od2 == 0.0) { // Stop divisions by zero
70             System.out.println("You can't divide by zero.");
71         } else {
72             System.out.println("Result:\t\t"+(od1/od2));
73         }
74         break;
75
76     case '*': // Multiply
77         System.out.println("Result:\t\t"+(od1*od2));
78         break;
79     }
80 } while(on != 'x');
81 }
82 }
```

Listing A.1: Original source code for the Calculator exercise and the 1st type of plagiarism.

A.2.2 Source code for the Calculator exercise, with the 2nd type of plagiarism.

The 2nd type of plagiarism (see Section 2.3.4) is when comments are changed.

```
1 // Import necessary Libraries
```

A.2. The program, Calculator

```
2 import java.util.Scanner;
3
4 class Calculator {
5     public static void main(String args[])
6     {
7         /* Operation and operands */
8         char on = ' ';
9         double od1 = 0.0, od2 = 0.0;
10
11         // Prepare a scanner to read the input
12         Scanner input = new Scanner(System.in);
13
14         // Show the program title
15         System.out.println("Calculator");
16
17         /* Repeat until exit is chosen */
18         do {
19             on = ' ';
20             od1 = 0.0;
21             od2 = 0.0;
22
23             // Show the Menu
24             System.out.println();
25             System.out.println("+ Add");
26             System.out.println("- Subtract");
27             System.out.println("* Multiply");
28             System.out.println("/ Divide");
29             System.out.println("x Exit");
30             System.out.println();
31
32             // Ask for the users choice
33             System.out.print("Choice:\t\t");
34
35             /* Read choice */
36             String temp;
37             try {
38                 temp = input.nextLine();
39                 on = temp.charAt(0);
40             } catch (Exception x) {
41                 on = ' ';
42             }
43
44             if (on == '+' || on == '-' || on == '/' || on == '*') {
45                 /* Get operands */
46                 System.out.print("Operand 1:\t");
47                 od1 = input.nextDouble();
48                 System.out.print("Operand 2:\t");
49                 od2 = input.nextDouble();
50                 input.nextLine();
51             } else if (on == 'x' || on == 'X') {
52                 // Do nothing if the user wants to exit
53             } else {
54                 // Give an error if its not a valid operation
55                 System.out.println("Unknown Operation!");
56             }
57         }
58     }
59 }
```

A.2. The program, Calculator

```
57
58     /* Do the math */
59     switch(on) {
60         case '+': // Add
61             System.out.println("Result:\t\t"+(od1+od2));
62             break;
63
64         case '-': // Subtract
65             System.out.println("Result:\t\t"+(od1-od2));
66             break;
67
68         case '/': // Divide
69             if(od2 == 0.0) { // Stop divisions by zero
70                 System.out.println("You can't divide by zero.");
71             } else {
72                 System.out.println("Result:\t\t"+(od1/od2));
73             }
74             break;
75
76         case '*': // Multiply
77             System.out.println("Result:\t\t"+(od1*od2));
78             break;
79     }
80 } while(on != 'x');
81 }
82 }
```

Listing A.2: Source code for the Calculator exercise with the 2nd type of plagiarism.

A.2.3 Source code for the Calculator exercise, with the 3rd type of plagiarism.

The 3rd type of plagiarism (see Section 2.3.4) is when identifiers (variable and function names) are changed.

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class SuperCalculator {
5     public static void main(String args[])
6     {
7         // Operation (on) and Operands (od)
8         char oper = ' ';
9         double first = 0.0, second = 0.0;
10
11         // Prepare a scanner to read the input
12         Scanner in = new Scanner(System.in);
13
14         // Show the program title
15         System.out.println("Calculator");
16     }
```

A.2. The program, Calculator

```
17 // Repeat until the user chooses to exit (x)
18 do {
19     oper = ' ';
20     first = 0.0;
21     second = 0.0;
22
23     // Show the Menu
24     System.out.println();
25     System.out.println("+ Add");
26     System.out.println("- Subtract");
27     System.out.println("* Multiply");
28     System.out.println("/ Divide");
29     System.out.println("x Exit");
30     System.out.println();
31
32     // Ask for the users choice
33     System.out.print("Choice:\t\t");
34
35     // Read the input (a single character)
36     String t;
37     try {
38         t = in.nextLine();
39         oper = \ch{t}.charAt(0);
40     } catch(Exception x) {
41         oper = ' ';
42     }
43
44     if(oper == '+' || oper == '-' || oper == '/' || oper == '*') {
45         // Ask for and read the operands if the user chose an operation
46         System.out.print("Operand 1:\t");
47         first = in.nextDouble();
48         System.out.print("Operand 2:\t");
49         second = in.nextDouble();
50         in.nextLine();
51     } else if(oper == 'x' || oper == 'X') {
52         // Do nothing if the user wants to exit
53     } else {
54         // Give an error if its not a valid operation
55         System.out.println("Unknown Operation!");
56     }
57
58     // Execute operation and report result
59     switch(oper) {
60         case '+': // Add
61             System.out.println("Result:\t\t"+(first+second));
62             break;
63
64         case '-': // Subtract
65             System.out.println("Result:\t\t"+(first-second));
66             break;
67
68         case '/': // Divide
69             if(second == 0.0) { // Stop divisions by zero
70                 System.out.println("You can't divide by zero.");
71             } else {
```

A.2. The program, Calculator

```
72         System.out.println("Result:\t\t"+(first/second));
73     }
74     break;
75
76     case '*': // Multiply
77         System.out.println("Result:\t\t"+(first*second));
78         break;
79     }
80 } while (oper != 'x');
81 }
82 }
```

Listing A.3: Source code for the Calculator exercise with the 3rd type of plagiarism.

A.2.4 Source code for the Calculator exercise, with the 4th type of plagiarism.

The 4th type of plagiarism (see Section 2.3.4) is when scopes are changed (making a local variable or function into a global one or vice versa).

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     // Operation (on) and Operands (od)
6     private static char on = ' ';
7     private static double od1 = 0.0, od2 = 0.0;
8
9     // Prepare a scanner to read the input
10    public static Scanner input = new Scanner(System.in);
11
12    public static void main(String args[])
13    {
14        // Operation (on) and Operands (od)
15        char on = ' ';
16        double od1 = 0.0, od2 = 0.0;
17
18        // Prepare a scanner to read the input
19        Scanner input = new Scanner(System.in);
20
21        // Show the program title
22        System.out.println("Calculator");
23
24        // Repeat until the user chooses to exit (x)
25        do {
26            on = ' ';
27            od1 = 0.0;
28            od2 = 0.0;
29
30            // Show the Menu
31            System.out.println();
```

A.2. The program, Calculator

```
32     System.out.println("+) Add");
33     System.out.println("-) Subtract");
34     System.out.println("*) Multiply");
35     System.out.println("/) Divide");
36     System.out.println("x) Exit");
37     System.out.println();
38
39     // Ask for the users choice
40     System.out.print("Choice:\t\t");
41
42     // Read the input (a single character)
43     String temp;
44     try {
45         temp = input.nextLine();
46         on = temp.charAt(0);
47     } catch(Exception x) {
48         on = ' ';
49     }
50
51     if(on == '+' || on == '-' || on == '/' || on == '*') {
52         // Ask for and read the operands if the user chose an operation
53         System.out.print("Operand 1:\t");
54         od1 = input.nextDouble();
55         System.out.print("Operand 2:\t");
56         od2 = input.nextDouble();
57         input.nextLine();
58     } else if(on == 'x' || on == 'X') {
59         // Do nothing if the user wants to exit
60     } else {
61         // Give an error if its not a valid operation
62         System.out.println("Unknown Operation!");
63     }
64
65     // Execute operation and report result
66     switch(on) {
67         case '+': // Add
68             System.out.println("Result:\t\t"+(od1+od2));
69             break;
70
71         case '-': // Subtract
72             System.out.println("Result:\t\t"+(od1-od2));
73             break;
74
75         case '/': // Divide
76             if(od2 == 0.0) { // Stop divisions by zero
77                 System.out.println("You can't divide by zero.");
78             } else {
79                 System.out.println("Result:\t\t"+(od1/od2));
80             }
81             break;
82
83         case '*': // Multiply
84             System.out.println("Result:\t\t"+(od1*od2));
85             break;
86     }
```

A.2. The program, Calculator

```
87     } while (on != 'x');
88   }
89 }
```

Listing A.4: Source code for the Calculator exercise with the 4th type of plagiarism.

A.2.5 Source code for the Calculator exercise, with the 5th type of plagiarism.

The 5th type of plagiarism (see Section 2.3.4) is when operands orders are switched (e.g. $x < y$ to $y >= x$).

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     public static void main(String args[])
6     {
7         // Operation (on) and Operands (od)
8         char on = ' ';
9         double od1 = 0.0, od2 = 0.0;
10
11         // Prepare a scanner to read the input
12         Scanner input = new Scanner(System.in);
13
14         // Show the program title
15         System.out.println("Calculator");
16
17         // Repeat until the user chooses to exit (x)
18         do {
19             on = ' ';
20             od1 = 0.0;
21             od2 = 0.0;
22
23             // Show the Menu
24             System.out.println();
25             System.out.println("+ Add");
26             System.out.println("- Subtract");
27             System.out.println("* Multiply");
28             System.out.println("/ Divide");
29             System.out.println("x Exit");
30             System.out.println();
31
32             // Ask for the users choice
33             System.out.print("Choice:\t\t");
34
35             // Read the input (a single character)
36             String temp;
37             try {
38                 temp = input.nextLine();
39                 on = temp.charAt(0);
```

A.2. The program, Calculator

```
40     } catch(Exception x) {
41         on = ' ';
42     }
43
44     if(on == '/' || on == '*' || on == '+' || on == '-') {
45         // Ask for and read the operands if the user chose an operation
46         System.out.print("Operand 1:\t");
47         od1 = input.nextDouble();
48         System.out.print("Operand 2:\t");
49         od2 = input.nextDouble();
50         input.nextLine();
51     } else if(on == 'X' || on == 'x') {
52         // Do nothing if the user wants to exit
53     } else {
54         // Give an error if its not a valid operation
55         System.out.println("Unknown Operation!");
56     }
57
58     // Execute operation and report result
59     switch(on) {
60         case '+': // Add
61             System.out.println("Result:\t\t"+(od2+od1));
62             break;
63
64         case '-': // Subtract
65             System.out.println("Result:\t\t"+(-od2+od1));
66             break;
67
68         case '/': // Divide
69             if(0.0 == od2) { // Stop divisions by zero
70                 System.out.println("You can't divide by zero.");
71             } else {
72                 System.out.println("Result:\t\t"+(od1/od2));
73             }
74             break;
75
76         case '*': // Multiply
77             System.out.println("Result:\t\t"+(od2*od1));
78             break;
79     }
80     } while ('x' != on);
81 }
82 }
```

Listing A.5: Source code for the Calculator exercise with the 5th type of plagiarism.

A.2. The program, Calculator

A.2.6 Source code for the Calculator exercise, with the 6th type of plagiarism.

The 6th type of plagiarism (see Section 2.3.4) is when variable types and control structures are replaced with equivalents (e.g. if else to switch case).

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     public static void main(String args[])
6     {
7         // Operation (on) and Operands (od)
8         String on = " ";
9         float od1 = 0.0f, od2 = 0.0f;
10
11         // Prepare a scanner to read the input
12         Scanner input = new Scanner(System.in);
13
14         // Show the program title
15         System.out.println("Calculator");
16
17         // Repeat until the user chooses to exit (x)
18         do {
19             on = " ";
20             od1 = 0.0f;
21             od2 = 0.0f;
22
23             // Show the Menu
24             System.out.println();
25             System.out.println("+ Add");
26             System.out.println("- Subtract");
27             System.out.println("* Multiply");
28             System.out.println("/ Divide");
29             System.out.println("x Exit");
30             System.out.println();
31
32             // Ask for the users choice
33             System.out.print("Choice:\t\t");
34
35             // Read the input (a single character)
36             String temp;
37             try {
38                 on = input.nextLine();
39                 on = temp.charAt(0);
40             } catch (Exception x) {
41                 on = " ";
42             }
43
44             if (on.charAt(0) == '+' || on.charAt(0) == '-' || on.charAt(0) == '/' || on.charAt(0)
45                 == '*') {
46                 // Ask for and read the operands if the user chose an operation
47                 System.out.print("Operand 1:\t");
48                 od1 = input.nextFloat();
```

A.2. The program, Calculator

```
48     System.out.print("Operand 2:\t");
49     od2 = input.nextFloat();
50     input.nextLine();
51 } else if(on.charAt(0) == 'x' || on.charAt(0) == 'X') {
52     // Do nothing if the user wants to exit
53 } else {
54     // Give an error if its not a valid operation
55     System.out.println("Unknown Operation!");
56 }
57
58 // Execute operation and report result
59 switch(on.charAt(0)) {
60     case '+': // Add
61         System.out.println("Result:\t\t"+(od1+od2));
62         break;
63
64     case '-': // Subtract
65         System.out.println("Result:\t\t"+(od1-od2));
66         break;
67
68     case '/': // Divide
69         if(od2 == 0.0f) { // Stop divisions by zero
70             System.out.println("You can't divide by zero.");
71         } else {
72             System.out.println("Result:\t\t"+(od1/od2));
73         }
74         break;
75
76     case '*': // Multiply
77         System.out.println("Result:\t\t"+(od1*od2));
78         break;
79     }
80 } while(on.charAt(0) != 'x');
81 }
82 }
```

Listing A.6: Source code for the Calculator exercise with the 6th type of plagiarism.

A.2.7 Source code for the Calculator exercise, with the 7th type of plagiarism.

The 7th type of plagiarism (see Section 2.3.4) is when statement orders are switched.

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     public static void main(String args[])
6     {
7         // Operation (on) and Operands (od)
8         double od1 = 0.0, od2 = 0.0;
```

A.2. The program, Calculator

```
9 char on = ' ';
10
11 // Show the program title
12 System.out.println("Calculator");
13
14 // Prepare a scanner to read the input
15 Scanner input = new Scanner(System.in);
16
17 // Repeat until the user chooses to exit (x)
18 do {
19     od1 = 0.0;
20     on = ' ';
21     od2 = 0.0;
22
23     // Show the Menu
24     System.out.println();
25     System.out.println("+ Add");
26     System.out.println("- Subtract");
27     System.out.println("* Multiply");
28     System.out.println("/ Divide");
29     System.out.println("x Exit");
30     System.out.println();
31
32     // Ask for the users choice
33     System.out.print("Choice:\t\t");
34
35     // Read the input (a single character)
36     String temp;
37     try {
38         temp = input.nextLine();
39         on = temp.charAt(0);
40     } catch (Exception x) {
41         on = ' ';
42     }
43
44     if (on == 'x' || on == 'X') {
45         // Do nothing if the user wants to exit
46     } else if (on == '+' || on == '-' || on == '/' || on == '*') {
47         // Ask for and read the operands if the user chose an operation
48         System.out.print("Operand 1:\t");
49         od1 = input.nextDouble();
50         System.out.print("Operand 2:\t");
51         od2 = input.nextDouble();
52         input.nextLine();
53     } else {
54         // Give an error if its not a valid operation
55         System.out.println("Unknown Operation!");
56     }
57
58     // Execute operation and report result
59     switch (on) {
60         case '/': // Divide
61             if (od2 == 0.0) { // Stop divisions by zero
62                 System.out.println("You can't divide by zero.");
63             } else {
```

A.2. The program, Calculator

```
64     System.out.println("Result:\t\t"+(od1/od2));
65     }
66     break;
67
68     case '*': // Multiply
69         System.out.println("Result:\t\t"+(od1*od2));
70         break;
71
72     case '+': // Add
73         System.out.println("Result:\t\t"+(od1+od2));
74         break;
75
76     case '-': // Subtract
77         System.out.println("Result:\t\t"+(od1-od2));
78         break;
79     }
80     } while(on != 'x');
81 }
82 }
```

Listing A.7: Source code for the Calculator exercise with the 7th type of plagiarism.

A.2.8 Source code for the Calculator exercise, with the 8th type of plagiarism.

The 8th type of plagiarism (see Section 2.3.4) is when groups of calls are turned into a function call or vice versa.

```
1 // Import necessary Libraries
2 import java.util.Scanner;
3
4 class Calculator {
5     public static double add(double op1, double op2) { return op1+op2; }
6     public static double sub(double op1, double op2) { return op1-op2; }
7     public static double mul(double op1, double op2) { return op1*op2; }
8     public static double div(double op1, double op2) { return op1/op2; }
9
10    public static char readChar() {
11        Scanner input = new Scanner(System.in);
12        String temp;
13        temp = input.nextLine();
14        return temp.charAt(0);
15    }
16
17    public static double readDouble() {
18        Scanner input = new Scanner(System.in);
19        double temp;
20        temp = input.nextDouble();
21        return temp;
22    }
23 }
```

A.2. The program, Calculator

```
24 public static void printMenu() {
25     System.out.println();
26     System.out.println("+) Add");
27     System.out.println("-) Subtract");
28     System.out.println("*) Multiply");
29     System.out.println("/) Divide");
30     System.out.println("x) Exit");
31     System.out.println();
32 }
33
34 public static void main(String args[])
35 {
36     // Operation (on) and Operands (od)
37     char on = ' ';
38     double od1 = 0.0, od2 = 0.0;
39
40     // Prepare a scanner to read the input
41     Scanner input = new Scanner(System.in);
42
43     // Show the program title
44     System.out.println("Calculator");
45
46     // Repeat until the user chooses to exit (x)
47     do {
48         on = ' ';
49         od1 = 0.0;
50         od2 = 0.0;
51
52         // Show the Menu
53         printMenu();
54
55         // Ask for the users choice
56         System.out.print("Choice:\t\t");
57
58         // Read the input (a single character)
59         on = readChar();
60
61         if(on == '+' || on == '-' || on == '/' || on == '*') {
62             // Ask for and read the operands if the user chose an operation
63             System.out.print("Operand 1:\t");
64             od1 = readDouble();
65             System.out.print("Operand 2:\t");
66             od2 = readDouble();
67         } else if(on == 'x' || on == 'X') {
68             // Do nothing if the user wants to exit
69         } else {
70             // Give an error if its not a valid operation
71             System.out.println("Unkownn Operation!");
72         }
73
74         // Execute operation and report result
75         switch(on) {
76             case '+': // Add
77                 System.out.println("Result:\t\t"+add(od1,od2));
78                 break;
```

A.3. The program, 21 Matches

```
79
80     case '-': // Subtract
81         System.out.println("Result:\t\t"+sub(od1,od2));
82         break;
83
84     case '/': // Divide
85         if(od2 == 0.0) { // Stop divisions by zero
86             System.out.println("You can't divide by zero.");
87         } else {
88             System.out.println("Result:\t\t"+div(od1,od2));
89         }
90         break;
91
92     case '*': // Multiply
93         System.out.println("Result:\t\t"+mul(od1,od2));
94         break;
95     }
96 } while(on != 'x');
97 }
98 }
```

Listing A.8: Source code for the Calculator exercise with the 8th type of plagiarism.

A.3 THE PROGRAM, 21 MATCHES

The objective was to make an interactive implementation of the 21 matches game where, starting with 21 matches, each player takes 1 to 4 matches until there are no more. The one getting the last match will lose the game. The game can be played against another player or against the computer.

The following metrics characterise the original file:

- **Number of lines:** 191
- **Number of functions:** 4
- **Number of variables:** 13

The C source code files are available online at https://tinyurl.com/dopisiae/source/21matches_c.zip, as well as the Java source code files, at https://tinyurl.com/dopisiae/source/21matches_java.zip.

A.3.1 *Original source code for the 21 Matches exercise and the 1st type of plagiarism.*

The 1st type of plagiarism (see Section 2.3.4) is an exact copy of the original.

A.3. The program, 21 Matches

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int CompTurn(int f, char e, int d)
5 { /* Function that handles a Computer turn and returns his choice */
6   /* Saves the number of matches that the Comp. will take */
7   int r = 1;
8
9   /* If its the Comp. turn, he uses the following algorithm */
10  if (e == 'n')
11  {
12    /* If all the matches are left, the Computer takes 4 matches */
13    if (f == 21)
14      r = 4;
15    else
16    {
17      if (f > 11)
18        r = 5-d + ((f-1) % 5);
19      else
20        r = 5-d;
21    }
22  }
23
24  /* Ensures that the Computer never takes more than 4 matches */
25  if (r > 4)
26    r = 4;
27
28  /* If it is not the First Player, use the following algorithm */
29  if (e == 'y')
30    r = 5-d;
31
32  /* If there are only 5 matches or less, the Computer makes the smart choice */
33  if ((f <= 5) && (f>1))
34    r = f-1;
35
36  /* Return the number of matches that the Comp. takes */
37  return r;
38 }
39
40 int PlayTurn(int f)
41 { /* Function that handles a Player turn and returns his choice */
42   /* Player Play */
43   int j = 1;
44
45   /* Is a valid Play (0 = No) */
46   int ok = 0;
47
48   do {
49     /* Asks how many Matches the Player wants to take ? */
50     printf("How many Matches will you take (1 to 4) : ");
51
52     /* Gets the number of Matches that the Player wants to take */
53     scanf(" %d",&j);
54
```

A.3. The program, 21 Matches

```
55     /* Verifies if the value is valid */
56     if(j < 1)
57         printf("\nNumber too Low!\n\n");
58     else if(j > 4)
59         printf("\nNumber too High!\n\n");
60     else if((f < 4) && (j > f))
61         printf("\nThere aren't that many matches left!\n\n");
62     else
63         ok = 1;
64 } while(ok == 0);
65
66 /* Returns the number of Matches that the Player takes */
67 return j;
68 }
69
70 void Game(int Type)
71 { /* Game Loop, ends when there are no matches left */
72     /* Says how many matches are left */
73     int f = 21;
74
75     /* Says if Game is versus Player or Computer */
76     char e = ' ';
77
78     /* Says whose turn it is (Player number) */
79     int v = 1;
80
81     /* Number of matches taken */
82     int d = 0;
83
84     /* On games versus Computer, ask who starts */
85     if(Type == 1)
86     {
87         do {
88             /* Ask and get the choice of the Player */
89             printf("\nDo you want to be the First Player (y or n) : ");
90             scanf(" %c",&e);
91
92             /* Warn the Player if the choice is invalid */
93             if ((e != 'y') && (e != 'n'))
94                 printf ("\nInvalid Choice !\n");
95         } while ((e != 'y') && (e != 'n'));
96     }
97
98     /* Main Game Loop */
99     while(f > 1)
100     {
101         /* Show matches left */
102         printf("\nMatches : %d\n",f);
103
104         /* Say whose turn it is */
105         printf("Player Turn : %d\n",v);
106
107         /* Who should play, according to Game type */
108         if(Type == 1)
109             { /* Hum. versus Comp. */
```


A.3. The program, 21 Matches

```
110     /* See who should play */
111     if ((e == 'n') && (v == 1) || (e == 'y') && (v != 1))
112     { /* Comp. turn */
113         /* Run the CompTurn function and get his play */
114         d = CompTurn(f,e,d);
115
116         /* Shows the question and the Comp. choice */
117         printf("How many matches do you want to take (1 to 4) : %d\n",d);
118     }
119     else
120     { /* Its the Hum. turn */
121         /* Run the PlayTurn function and get the Play. choice */
122         d = PlayTurn(f);
123     }
124 }
125 else
126 { /* Human versus Human */
127     /* Run the PlayTurn function and get the Player choice */
128     d = PlayTurn(f);
129 }
130
131 /* Take matches from the total */
132 f -= d;
133
134 /* If no one won, Go to next turn */
135 if(f > 1)
136 {
137     v++;
138     if (v > 2)
139         v = 1;
140 }
141 }
142
143 /* Show who Won */
144 printf("\nThe Winner is Player %d!\n", v);
145 }
146
147 int main()
148 { /* Main Loop,the Menu is a Loop that only stops when the User asks to Exit */
149     /* Player Option */
150     int op = 0;
151
152     /* Presentation of the Game and its Rules */
153     printf("Game of the 21 Matches\n");
154     printf("\nRules:\n");
155     printf("\tThere are 21 Matches.\n");
156     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
157     printf("\tThe Player that takes the last Match loses.\n");
158
159     /* Beginning of the Main Loop */
160     do {
161         /* Initial Menu */
162         printf("\n1) Play, Human versus Computer\n");
163         printf("\n2) Play, Human versus Human\n");
164         printf("\n0) Exit\n\n");
```

A.3. The program, 21 Matches

```
165
166     /* Get and Interpret the Choice */
167     printf("Choice: ");
168     scanf(" %d",&op);
169
170     switch(op)
171     {
172     case 0:
173         /* Exit the Game */
174         break;
175     case 1:
176         /* Run the Game */
177         Game(op);
178         break;
179     case 2:
180         /* Run the Game */
181         Game(op);
182         break;
183     default:
184         /* Tell the User that he made an Invalid Choice */
185         printf("\nInvalid Number !\n");
186     }
187 } while(op != 0);
188
189 /* End of the Loop and the Game */
190 return 0;
191 }
```

Listing A.9: Original source code for the 21 Matches exercise and the 1st type of plagiarism.

A.3.2 Source code for the 21 Matches exercise, with the 2nd type of plagiarism.

The 2nd type of plagiarism (see Section 2.3.4) is when comments are changed.

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int CompTurn(int f, char e, int d)
5 { /* Function that handles a Computer turn and returns his choice */
6     /* Number of matches that the Comp. will take */
7     int r = 1;
8
9     /* Computer Turn */
10    if (e == 'n')
11    {
12        /* If all the matches are left, the Computer takes 4 matches */
13        if (f == 21)
14            r = 4;
15        else
16        {
```

A.3. The program, 21 Matches

```
17     if (f > 11)
18         r = 5-d + ((f-1) % 5);
19     else
20         r = 5-d;
21     }
22 }
23
24 /* Never take more than 4 matches */
25 if (r > 4)
26     r = 4;
27
28 /* If it is not the First Player, use the following algorithm */
29 if (e == 'y')
30     r = 5-d;
31
32 /* If there are only 5 matches or less, the Computer makes the smart choice */
33 if ((f <= 5) && (f>1))
34     r = f-1;
35
36 /* Return the number of matches taken */
37 return r;
38 }
39
40 int PlayTurn(int f)
41 { /* Function that handles a Player turn and returns his choice */
42     /* Number of matches that the Play. will take */
43     int j = 1;
44
45     /* Is the play valid ? */
46     int ok = 0;
47
48     do {
49         printf("How many Matches will you take (1 to 4) : ");
50
51         scanf(" %d",&j);
52
53         /* Control matches taken */
54         if(j < 1)
55             printf("\nNumber too Low!\n\n");
56         else if(j > 4)
57             printf("\nNumber too High!\n\n");
58         else if((f < 4) && (j > f))
59             printf("\nThere aren't that many matches left!\n\n");
60         else
61             ok = 1;
62     } while(ok == 0);
63
64     /* Returns the number of Matches that the Player takes */
65     return j;
66 }
67
68 void Game(int Type)
69 { /* Game Loop, ends when there are no matches left */
70     /* Number of matches left */
71     int f = 21;
```

A.3. The program, 21 Matches

```
72
73 /* Type of game */
74 char e = ' ';
75
76 /* Whose turn is it ? */
77 int v = 1;
78
79 /* Number of matches taken */
80 int d = 0;
81
82 /* On games versus Computer, ask who starts */
83 if(Type == 1)
84 {
85     do {
86         /* Ask and get the choice of the Player */
87         printf("\nDo you want to be the First Player (y or n) : ");
88         scanf(" %c",&e);
89
90         /* Warn the Player if the choice is invalid */
91         if ((e != 'y') && (e != 'n'))
92             printf ("\nInvalid Choice !\n");
93     } while ((e != 'y') && (e != 'n'));
94 }
95
96 /* Main Game Loop */
97 while(f > 1)
98 {
99     /* Show matches left */
100    printf("\nMatches : %d\n",f);
101
102    /* Say whose turn it is */
103    printf("Player Turn : %d\n",v);
104
105    /* Who should play, according to Game type */
106    if(Type == 1)
107    { /* Hum. versus Comp. */
108        /* Who plays ? */
109        if (((e == 'n') && (v == 1)) || ((e == 'y') && (v != 1)))
110        { /* Comp. turn */
111            /* Run the CompTurn function and get his play */
112            d = CompTurn(f,e,d);
113
114            /* Shows the question and the Comp. choice */
115            printf("How many matches do you want to take (1 to 4) : %d\n",d);
116        }
117        else
118        { /* Its the Hum. turn */
119            /* Run the PlayTurn function and get the Play. choice */
120            d = PlayTurn(f);
121        }
122    }
123    else
124    { /* Human versus Human */
125        /* Run the PlayTurn function and get the Player choice */
126        d = PlayTurn(f);
```

A.3. The program, 21 Matches

```
127     }
128
129     /* Take matches from the total */
130     f -= d;
131
132     /* If no one won, repeat */
133     if(f > 1)
134     {
135         v++;
136         if (v > 2)
137             v = 1;
138     }
139 }
140
141 /* Show who Won */
142 printf("\nThe Winner is Player %d!\n", v);
143 }
144
145 int main()
146 { /* Main Loop, the Menu is a Loop that only stops when the User asks to Exit */
147     /* Player Option */
148     int op = 0;
149
150     /* Presentation of the Game and its Rules */
151     printf("Game of the 21 Matches\n");
152     printf("\nRules:\n");
153     printf("\tThere are 21 Matches.\n");
154     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
155     printf("\tThe Player that takes the last Match loses.\n");
156
157     /* Beginning of the Main Loop */
158     do {
159         /* Initial Menu */
160         printf("\n1) Play, Human versus Computer\n");
161         printf("2) Play, Human versus Human\n");
162         printf("0) Exit\n\n");
163
164         /* Get and Interpret the Choice */
165         printf("Choice: ");
166         scanf(" %d",&op);
167
168         switch(op)
169         {
170         case 0:
171             /* Quit game */
172             break;
173         case 1:
174             /* Human versus Computer game */
175             Game(op);
176             break;
177         case 2:
178             /* Human versus Human game */
179             Game(op);
180             break;
181         default:
```

A.3. The program, 21 Matches

```
182     /* Tell the User that he made an Invalid Choice */
183     printf("\nInvalid Number !\n");
184 }
185 } while(op != 0);
186
187 /* End of the Loop and the Game */
188 return 0;
189 }
```

Listing A.10: Source code for the 21 Matches exercise with the 2nd type of plagiarism.

A.3.3 Source code for the 21 Matches exercise, with the 3rd type of plagiarism.

The 3rd type of plagiarism (see Section 2.3.4) is when identifiers (variable and function names) are changed.

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int CoTu(int left, char first, int taken)
5 { /* Function that handles a Computer turn and returns his choice */
6     /* Saves the number of matches that the Comp. will take */
7     int play = 1;
8
9     /* If its the Comp. turn, he uses the following algorithm */
10    if (first == 'n')
11    {
12        /* If all the matches are left, the Computer takes 4 matches */
13        if (left == 21)
14            play = 4;
15        else
16        {
17            if (left > 11)
18                play = 5-taken + ((left-1) % 5);
19            else
20                play = 5-taken;
21        }
22    }
23
24    /* Ensures that the Computer never takes more than 4 matches */
25    if (play > 4)
26        play = 4;
27
28    /* If it is not the First Player, use the following algorithm */
29    if (first == 'y')
30        play = 5-taken;
31
32    /* If there are only 5 matches or less, the Computer makes the smart choice */
33    if ((left <= 5) && (left>1))
34        play = left-1;
```

A.3. The program, 21 Matches

```
35
36  /* Return the number of matches that the Comp. takes */
37  return play;
38 }
39
40 int PlTu(int left)
41 { /* Function that handles a Player turn and returns his choice */
42  /* Player Play */
43  int play = 1;
44
45  /* Is a valid Play (0 = No) */
46  int ok = 0;
47
48  do {
49    /* Asks how many Matches the Player wants to take ? */
50    printf("How many Matches will you take (1 to 4) : ");
51
52    /* Gets the number of Matches that the Player wants to take */
53    scanf(" %d",&play);
54
55    /* Verifies if the value is valid */
56    if(play < 1)
57        printf("\nNumber too Low!\n\n");
58    else if(play > 4)
59        printf("\nNumber too High!\n\n");
60    else if((left < 4) && (play > left))
61        printf("\nThere aren't that many matches left!\n\n");
62    else
63        ok = 1;
64  } while(ok == 0);
65
66  /* Returns the number of Matches that the Player takes */
67  return play;
68 }
69
70 void Ga(int t)
71 { /* Game Loop, ends when there are no matches left */
72  /* Says how many matches are left */
73  int left = 21;
74
75  /* Says if Game is versus Player or Computer */
76  char first = ' ';
77
78  /* Says whose turn it is (Player number) */
79  int turn = 1;
80
81  /* Number of matches taken */
82  int taken = 0;
83
84  /* On games versus Computer, ask who starts */
85  if(t == 1)
86  {
87      do {
88          /* Ask and get the choice of the Player */
89          printf("\nDo you want to be the First Player (y or n) : ");
```

A.3. The program, 21 Matches

```
90     scanf(" %c",&first);
91
92     /* Warn the Player if the choice is invalid */
93     if ((first != 'y') && (first != 'n'))
94         printf ("\nInvalid Choice !\n");
95     } while ((first != 'y') && (first != 'n'));
96 }
97
98 /* Main Game Loop */
99 while(left > 1)
100 {
101     /* Show matches left */
102     printf("\nMatches : %d\n",left);
103
104     /* Say whose turn it is */
105     printf("Player Turn : %d\n",turn);
106
107     /* Who should play, according to Game type */
108     if(t == 1)
109     { /* Hum. versus Comp. */
110         /* See who should play */
111         if (((first == 'n') && (turn == 1)) || ((first == 'y') && (turn != 1)))
112         { /* Comp. turn */
113             /* Run the CompTurn function and get his play */
114             taken = CoTu(left,first,taken);
115
116             /* Shows the question and the Comp. choice */
117             printf("How many matches do you want to take (1 to 4) : %d\n",taken);
118         }
119         else
120         { /* Its the Hum. turn */
121             /* Run the PlayTurn function and get the Play. choice */
122             taken = PlTu(left);
123         }
124     }
125     else
126     { /* Human versus Human */
127         /* Run the PlayTurn function and get the Player choice */
128         taken = PlTu(left);
129     }
130
131     /* Take matches from the total */
132     left -= taken;
133
134     /* If no one won, Go to next turn */
135     if(left > 1)
136     {
137         turn++;
138         if (turn > 2)
139             turn = 1;
140     }
141 }
142
143 /* Show who Won */
144 printf("\nThe Winner is Player %d!\n", turn);
```


A.3. The program, 21 Matches

```
145 }
146
147 int main()
148 { /* Main Loop,the Menu is a Loop that only stops when the User asks to Exit */
149 /* Player Option */
150 int op = 0;
151
152 /* Presentation of the Game and its Rules */
153 printf("Game of the 21 Matches\n");
154 printf("\nRules:\n");
155 printf("\tThere are 21 Matches.\n");
156 printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
157 printf("\tThe Player that takes the last Match loses.\n");
158
159 /* Beginning of the Main Loop */
160 do {
161 /* Initial Menu */
162 printf("\n1) Play, Human versus Computer\n");
163 printf("2) Play, Human versus Human\n");
164 printf("0) Exit\n\n");
165
166 /* Get and Interpret the Choice */
167 printf("Choice: ");
168 scanf(" %d",&op);
169
170 switch(op)
171 {
172 case 0:
173 /* Exit the Game */
174 break;
175 case 1:
176 /* Run the Game */
177 Ga(op);
178 break;
179 case 2:
180 /* Run the Game */
181 Ga(op);
182 break;
183 default:
184 /* Tell the User that he made an Invalid Choice */
185 printf("\nInvalid Number !\n");
186 }
187 } while(op != 0);
188
189 /* End of the Loop and the Game */
190 return 0;
191 }
```

Listing A.11: Source code for the 21 Matches exercise with the 3rd type of plagiarism.

A.3. The program, 21 Matches

A.3.4 Source code for the 21 Matches exercise, with the 4th type of plagiarism.

The 4th type of plagiarism (see Section 2.3.4) is when scopes are changed (making a local variable or function into a global one or vice versa).

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 /* Says how many matches are left */
5 int f = 21;
6
7 /* Says if Game is versus Player or Computer */
8 char e = ' ';
9
10 /* Says whose turn it is (Player number) */
11 int v = 1;
12
13 /* Number of matches taken */
14 int d = 0;
15
16 int CompTurn(int f, char e, int d)
17 { /* Function that handles a Computer turn and returns his choice */
18   /* Saves the number of matches that the Comp. will take */
19   int r = 1;
20
21   /* If its the Comp. turn, he uses the following algorithm */
22   if (e == 'n')
23   {
24     /* If all the matches are left, the Computer takes 4 matches */
25     if (f == 21)
26       r = 4;
27     else
28     {
29       if (f > 11)
30         r = 5-d + ((f-1) % 5);
31       else
32         r = 5-d;
33     }
34   }
35
36   /* Ensures that the Computer never takes more than 4 matches */
37   if (r > 4)
38     r = 4;
39
40   /* If it is not the First Player, use the following algorithm */
41   if (e == 'y')
42     r = 5-d;
43
44   /* If there are only 5 matches or less, the Computer makes the smart choice */
45   if ((f <= 5) && (f>1))
46     r = f-1;
47
48   /* Return the number of matches that the Comp. takes */
```

A.3. The program, 21 Matches

```
49  return r;
50  }
51
52  int PlayTurn(int f)
53  { /* Function that handles a Player turn and returns his choice */
54    /* Player Play */
55    int j = 1;
56
57    /* Is a valid Play (0 = No) */
58    int ok = 0;
59
60    do {
61      /* Asks how many Matches the Player wants to take ? */
62      printf("How many Matches will you take (1 to 4) : ");
63
64      /* Gets the number of Matches that the Player wants to take */
65      scanf(" %d",&j);
66
67      /* Verifies if the value is valid */
68      if(j < 1)
69        printf("\nNumber too Low!\n\n");
70      else if(j > 4)
71        printf("\nNumber too High!\n\n");
72      else if((f < 4) && (j > f))
73        printf("\nThere aren't that many matches left!\n\n");
74      else
75        ok = 1;
76    } while(ok == 0);
77
78    /* Returns the number of Matches that the Player takes */
79    return j;
80  }
81
82  void Game(int Type)
83  { /* Game Loop, ends when there are no matches left */
84    /* Says how many matches are left */
85    int f = 21;
86
87    /* Says if Game is versus Player or Computer */
88    char e = ' ';
89
90    /* Says whose turn it is (Player number) */
91    int v = 1;
92
93    /* Number of matches taken */
94    int d = 0;
95
96    f = 21;
97    e = ' ';
98    v = 1;
99    d = 0;
100
101    /* On games versus Computer, ask who starts */
102    if(Type == 1)
103    {
```

A.3. The program, 21 Matches

```
104 do {
105     /* Ask and get the choice of the Player */
106     printf("\nDo you want to be the First Player (y or n) : ");
107     scanf(" %c",&e);
108
109     /* Warn the Player if the choice is invalid */
110     if ((e != 'y') && (e != 'n'))
111         printf("\nInvalid Choice !\n");
112 } while ((e != 'y') && (e != 'n'));
113 }
114
115 /* Main Game Loop */
116 while(f > 1)
117 {
118     /* Show matches left */
119     printf("\nMatches : %d\n",f);
120
121     /* Say whose turn it is */
122     printf("Player Turn : %d\n",v);
123
124     /* Who should play, according to Game type */
125     if(Type == 1)
126     { /* Hum. versus Comp. */
127         /* See who should play */
128         if (((e == 'n') && (v == 1)) || ((e == 'y') && (v != 1)))
129         { /* Comp. turn */
130             /* Run the CompTurn function and get his play */
131             d = CompTurn(f,e,d);
132
133             /* Shows the question and the Comp. choice */
134             printf("How many matches do you want to take (1 to 4) : %d\n",d);
135         }
136     }
137     else
138     { /* Its the Hum. turn */
139         /* Run the PlayTurn function and get the Play. choice */
140         d = PlayTurn(f);
141     }
142 }
143 else
144 { /* Human versus Human */
145     /* Run the PlayTurn function and get the Player choice */
146     d = PlayTurn(f);
147 }
148
149 /* Take matches from the total */
150 f -= d;
151
152 /* If no one won, Go to next turn */
153 if(f > 1)
154 {
155     v++;
156     if (v > 2)
157         v = 1;
158 }
```

A.3. The program, 21 Matches

```
159
160 /* Show who Won */
161 printf("\nThe Winner is Player %d!\n", v);
162 }
163
164 int main()
165 { /* Main Loop, the Menu is a Loop that only stops when the User asks to Exit */
166 /* Player Option */
167 int op = 0;
168
169 /* Presentation of the Game and its Rules */
170 printf("Game of the 21 Matches\n");
171 printf("\nRules:\n");
172 printf("\t\tThere are 21 Matches.\n");
173 printf("\t\tEach Player can take 1 to 4 Matches in his turn.\n");
174 printf("\t\tThe Player that takes the last Match loses.\n");
175
176 /* Beginning of the Main Loop */
177 do {
178 /* Initial Menu */
179 printf("\n1) Play, Human versus Computer\n");
180 printf("2) Play, Human versus Human\n");
181 printf("0) Exit\n\n");
182
183 /* Get and Interpret the Choice */
184 printf("Choice: ");
185 scanf(" %d",&op);
186
187 switch(op)
188 {
189 case 0:
190 /* Exit the Game */
191 break;
192 case 1:
193 /* Run the Game */
194 Game(op);
195 break;
196 case 2:
197 /* Run the Game */
198 Game(op);
199 break;
200 default:
201 /* Tell the User that he made an Invalid Choice */
202 printf("\nInvalid Number !\n");
203 }
204 } while(op != 0);
205
206 /* End of the Loop and the Game */
207 return 0;
208 }
```

Listing A.12: Source code for the 21 Matches exercise with the 4th type of plagiarism.

A.3. The program, 21 Matches

A.3.5 Source code for the 21 Matches exercise, with the 5th type of plagiarism.

The 5th type of plagiarism (see Section 2.3.4) is when operands orders are switched (e.g. $x < y$ to $y >= x$).

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int CompTurn(int f, char e, int d)
5 { /* Function that handles a Computer turn and returns his choice */
6   /* Saves the number of matches that the Comp. will take */
7   int r = 1;
8
9   /* If its the Comp. turn, he uses the following algorithm */
10  if ('n' == e)
11  {
12    /* If all the matches are left, the Computer takes 4 matches */
13    if (21 == f)
14      r = 4;
15    else
16    {
17      if (11 > f)
18        r = ((-1+f) % 5) + -d+5;
19      else
20        r = -d+5;
21    }
22  }
23
24  /* Ensures that the Computer never takes more than 4 matches */
25  if (4 < r)
26    r = 4;
27
28  /* If it is not the First Player, use the following algorithm */
29  if ('y' == e)
30    r = -d+5;
31
32  /* If there are only 5 matches or less, the Computer makes the smart choice */
33  if ((5 >= f) && (1>f))
34    r = -1+f;
35
36  /* Return the number of matches that the Comp. takes */
37  return r;
38 }
39
40 int PlayTurn(int f)
41 { /* Function that handles a Player turn and returns his choice */
42   /* Player Play */
43   int j = 1;
44
45   /* Is a valid Play (0 = No) */
46   int ok = 0;
47
48   do {
```

A.3. The program, 21 Matches

```
49     /* Asks how many Matches the Player wants to take ? */
50     printf("How many Matches will you take (1 to 4) : ");
51
52     /* Gets the number of Matches that the Player wants to take */
53     scanf(" %d",&j);
54
55     /* Verifies if the value is valid */
56     if(1 > j)
57         printf("\nNumber too Low!\n\n");
58     else if(4 < j)
59         printf("\nNumber too High!\n\n");
60     else if((f < j) && (4 > f))
61         printf("\nThere aren't that many matches left!\n\n");
62     else
63         ok = 1;
64 } while(ok == 0);
65
66 /* Returns the number of Matches that the Player takes */
67 return j;
68 }
69
70 void Game(int Type)
71 { /* Game Loop, ends when there are no matches left */
72     /* Says how many matches are left */
73     int f = 21;
74
75     /* Says if Game is versus Player or Computer */
76     char e = ' ';
77
78     /* Says whose turn it is (Player number) */
79     int v = 1;
80
81     /* Number of matches taken */
82     int d = 0;
83
84     /* On games versus Computer, ask who starts */
85     if(1 == Type)
86     {
87         do {
88             /* Ask and get the choice of the Player */
89             printf("\nDo you want to be the First Player (y or n) : ");
90             scanf(" %c",&e);
91
92             /* Warn the Player if the choice is invalid */
93             if (('n' != e) && ('y' != e))
94                 printf (" \nInvalid Choice !\n");
95             } while (('n' != e) && ('y' != e));
96     }
97
98     /* Main Game Loop */
99     while(1 < f)
100     {
101         /* Show matches left */
102         printf("\nMatches : %d\n",f);
103
```

A.3. The program, 21 Matches

```
104     /* Say whose turn it is */
105     printf("Player Turn : %d\n",v);
106
107     /* Who should play, according to Game type */
108     if(l == Type)
109     { /* Hum. versus Comp. */
110         /* See who should play */
111         if (((l == v) && ('n' == e)) || ((l != v) && ('y' == e)))
112         { /* Comp. turn */
113             /* Run the CompTurn function and get his play */
114             d = CompTurn(f,e,d);
115
116             /* Shows the question and the Comp. choice */
117             printf("How many matches do you want to take (1 to 4) : %d\n",d);
118         }
119     else
120     { /* Its the Hum. turn */
121         /* Run the PlayTurn function and get the Play. choice */
122         d = PlayTurn(f);
123     }
124 }
125 else
126 { /* Human versus Human */
127     /* Run the PlayTurn function and get the Player choice */
128     d = PlayTurn(f);
129 }
130
131 /* Take matches from the total */
132 f -= d;
133
134 /* If no one won, Go to next turn */
135 if(l < f)
136 {
137     v++;
138     if (2 < v)
139         v = 1;
140 }
141 }
142
143 /* Show who Won */
144 printf("\nThe Winner is Player %d!\n", v);
145 }
146
147 int main()
148 { /* Main Loop,the Menu is a Loop that only stops when the User asks to Exit */
149     /* Player Option */
150     int op = 0;
151
152     /* Presentation of the Game and its Rules */
153     printf("Game of the 21 Matches\n");
154     printf("\nRules:\n");
155     printf("\tThere are 21 Matches.\n");
156     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
157     printf("\tThe Player that takes the last Match loses.\n");
158 }
```


A.3. The program, 21 Matches

```
159  /* Beginning of the Main Loop */
160  do {
161      /* Initial Menu */
162      printf("\n1) Play, Human versus Computer\n");
163      printf("2) Play, Human versus Human\n");
164      printf("0) Exit\n\n");
165
166      /* Get and Interpret the Choice */
167      printf("Choice: ");
168      scanf(" %d",&op);
169
170      switch(op)
171      {
172      case 0:
173          /* Exit the Game */
174          break;
175      case 1:
176          /* Run the Game */
177          Game(op);
178          break;
179      case 2:
180          /* Run the Game */
181          Game(op);
182          break;
183      default:
184          /* Tell the User that he made an Invalid Choice */
185          printf("\nInvalid Number !\n");
186      }
187  } while(0 != op);
188
189  /* End of the Loop and the Game */
190  return 0;
191 }
```

Listing A.13: Source code for the 21 Matches exercise with the 5th type of plagiarism.

A.3.6 Source code for the 21 Matches exercise, with the 6th type of plagiarism.

The 6th type of plagiarism (see Section 2.3.4) is when variable types and control structures are replaced with equivalents (e.g. if else to switch case).

```
1  /* Library(s) used by the Program */
2  #include <stdio.h>
3
4  char CompTurn(char f, char e, char d)
5  { /* Function that handles a Computer turn and returns his choice */
6      /* Saves the number of matches that the Comp. will take */
7      char r = 1;
8
9      /* If its the Comp. turn, he uses the following algorithm */
```

A.3. The program, 21 Matches

```
10  if (e == 'n')
11  {
12      /* If all the matches are left, the Computer takes 4 matches */
13      if (f == 21)
14          r = 4;
15      else
16      {
17          if (f > 11)
18              r = 5-d + ((f-1) % 5);
19          else
20              r = 5-d;
21      }
22  }
23
24  /* Ensures that the Computer never takes more than 4 matches */
25  if (r > 4)
26      r = 4;
27
28  /* If it is not the First Player, use the following algorithm */
29  if (e == 'y')
30      r = 5-d;
31
32  /* If there are only 5 matches or less, the Computer makes the smart choice */
33  if ((f <= 5) && (f>1))
34      r = f-1;
35
36  /* Return the number of matches that the Comp. takes */
37  return r;
38  }
39
40  char PlayTurn(char f)
41  { /* Function that handles a Player turn and returns his choice */
42      /* Player Play */
43      char j = 1;
44
45      /* Is a valid Play (0 = No) */
46      char ok = 0;
47
48      do {
49          /* Asks how many Matches the Player wants to take ? */
50          printf("How many Matches will you take (1 to 4) : ");
51
52          /* Gets the number of Matches that the Player wants to take */
53          scanf(" %c",&j);
54          j -= '0';
55
56          /* Verifies if the value is valid */
57          if(j < 1)
58              printf("\nNumber too Low!\n\n");
59          else if(j > 4)
60              printf("\nNumber too High!\n\n");
61          else if((f < 4) && (j > f))
62              printf("\nThere aren't that many matches left!\n\n");
63          else
64              ok = 1;
```

A.3. The program, 21 Matches

```
65 } while(ok == 0);
66
67 /* Returns the number of Matches that the Player takes */
68 return j;
69 }
70
71 void Game(char Type)
72 { /* Game Loop, ends when there are no matches left */
73   /* Says how many matches are left */
74   char f = 21;
75
76   /* Says if Game is versus Player or Computer */
77   char e = ' ';
78
79   /* Says whose turn it is (Player number) */
80   char v = 1;
81
82   /* Number of matches taken */
83   char d = 0;
84
85   /* On games versus Computer, ask who starts */
86   if(Type == '1')
87   {
88     do {
89       /* Ask and get the choice of the Player */
90       printf("\nDo you want to be the First Player (y or n) : ");
91       scanf(" %c",&e);
92
93       /* Warn the Player if the choice is invalid */
94       if ((e != 'y') && (e != 'n'))
95         printf ("\nInvalid Choice !\n");
96     } while ((e != 'y') && (e != 'n'));
97   }
98
99   /* Main Game Loop */
100  while(f > 1)
101  {
102    /* Show matches left */
103    printf("\nMatches : %d\n",f);
104
105    /* Say whose turn it is */
106    printf("Player Turn : %d\n",v);
107
108    /* Who should play, according to Game type */
109    if(Type == '1')
110    { /* Hum. versus Comp. */
111      /* See who should play */
112      if (((e == 'n') && (v == 1)) || ((e == 'y') && (v != 1)))
113      { /* Comp. turn */
114        /* Run the CompTurn function and get his play */
115        d = CompTurn(f,e,d);
116
117        /* Shows the question and the Comp. choice */
118        printf("How many matches do you want to take (1 to 4) : %d\n",d);
119      }

```

A.3. The program, 21 Matches

```
120     else
121     { /* Its the Hum. turn */
122         /* Run the PlayTurn function and get the Play. choice */
123         d = PlayTurn(f);
124     }
125 }
126 else
127 { /* Human versus Human */
128     /* Run the PlayTurn function and get the Player choice */
129     d = PlayTurn(f);
130 }
131
132 /* Take matches from the total */
133 f -= (int) d;
134
135 /* If no one won, Go to next turn */
136 if(f > 1)
137 {
138     v++;
139     if (v > 2)
140         v = 1;
141 }
142 }
143
144 /* Show who Won */
145 printf("\nThe Winner is Player %d!\n", (int) v);
146 }
147
148 int main()
149 { /* Main Loop, the Menu is a Loop that only stops when the User asks to Exit */
150     /* Player Option */
151     char op = 0;
152
153     /* Presentation of the Game and its Rules */
154     printf("Game of the 21 Matches\n");
155     printf("\nRules:\n");
156     printf("\tThere are 21 Matches.\n");
157     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
158     printf("\tThe Player that takes the last Match loses.\n");
159
160     /* Beginning of the Main Loop */
161     do {
162         /* Initial Menu */
163         printf("\n1) Play, Human versus Computer\n");
164         printf("2) Play, Human versus Human\n");
165         printf("0) Exit\n\n");
166
167         /* Get and Interpret the Choice */
168         printf("Choice: ");
169         scanf(" %c", &op);
170
171         if(op == '0')
172         {
173             case '0':
174                 /* Exit the Game */
```

A.3. The program, 21 Matches

```
175     break;
176 }
177 else if (op == '1')
178 {
179     case '1':
180         /* Run the Game */
181         Game(op);
182         break;
183 }
184 else if (op == '2')
185 {
186     case '2':
187         /* Run the Game */
188         Game(op);
189         break;
190 }
191 else
192 {
193     default:
194         /* Tell the User that he made an Invalid Choice */
195         printf("\nInvalid Number !\n");
196     }
197 } while (op != '0');
198
199 /* End of the Loop and the Game */
200 return 0;
201 }
```

Listing A.14: Source code for the 21 Matches exercise with the 6th type of plagiarism.

A.3.7 Source code for the 21 Matches exercise, with the 7th type of plagiarism.

The 7th type of plagiarism (see Section 2.3.4) is when statement orders are switched.

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int PlayTurn(int f)
5 { /* Function that handles a Player turn and returns his choice */
6     /* Is a valid Play (0 = No) */
7     int ok = 0;
8
9     /* Player Play */
10    int j = 1;
11
12    do {
13        /* Asks how many Matches the Player wants to take ? */
14        printf("How many Matches will you take (1 to 4) : ");
15
16        /* Gets the number of Matches that the Player wants to take */
```

A.3. The program, 21 Matches

```
17     scanf(" %d",&j);
18
19     /* Verifies if the value is valid */
20     if((f < 4) && (j > f))
21         printf("\nThere aren't that many matches left!\n\n");
22     else if(j > 4)
23         printf("\nNumber too High!\n\n");
24     else if(j < 1)
25         printf("\nNumber too Low!\n\n");
26     else
27         ok = 1;
28 } while(ok == 0);
29
30 /* Returns the number of Matches that the Player takes */
31 return j;
32 }
33
34 int CompTurn(int f, char e, int d)
35 { /* Function that handles a Computer turn and returns his choice */
36     /* Saves the number of matches that the Comp. will take */
37     int r = 1;
38
39     /* If its the Comp. turn, he uses the following algorithm */
40     if (e == 'n')
41     {
42         /* If all the matches are left, the Computer takes 4 matches */
43         if (f != 21)
44         {
45             if (f > 11)
46                 r = 5-d + ((f-1) % 5);
47             else
48                 r = 5-d;
49         }
50     else
51         r = 4;
52     }
53
54     /* Ensures that the Computer never takes more than 4 matches */
55     if (r > 4)
56         r = 4;
57
58     /* If it is not the First Player, use the following algorithm */
59     if (e == 'y')
60         r = 5-d;
61
62     /* If there are only 5 matches or less, the Computer makes the smart choice */
63     if ((f <= 5) && (f>1))
64         r = f-1;
65
66     /* Return the number of matches that the Comp. takes */
67     return r;
68 }
69
70 void Game(int Type)
71 { /* Game Loop, ends when there are no matches left */
```

A.3. The program, 21 Matches

```
72  /* Says whose turn it is (Player number) */
73  int v = 1;
74
75  /* Number of matches taken */
76  int d = 0;
77
78  /* Says how many matches are left */
79  int f = 21;
80
81  /* Says if Game is versus Player or Computer */
82  char e = ' ';
83
84  /* On games versus Computer, ask who starts */
85  if(Type == 1)
86  {
87      do {
88          /* Ask and get the choice of the Player */
89          printf("\nDo you want to be the First Player (y or n) : ");
90          scanf(" %c",&e);
91
92          /* Warn the Player if the choice is invalid */
93          if ((e != 'y') && (e != 'n'))
94              printf ("\nInvalid Choice !\n");
95      } while ((e != 'y') && (e != 'n'));
96  }
97
98  /* Main Game Loop */
99  while(f > 1)
100  {
101      /* Show matches left */
102      printf("\nMatches : %d\n",f);
103
104      /* Say whose turn it is */
105      printf("Player Turn : %d\n",v);
106
107      /* Who should play, according to Game type */
108      if(Type == 1)
109      { /* Hum. versus Comp. */
110          /* See who should play */
111          if (!( ((e == 'n') && (v == 1)) || ((e == 'y') && (v != 1))))
112          { /* Its the Hum. turn */
113              /* Run the PlayTurn function and get the Play. choice */
114              d = PlayTurn(f);
115          }
116          else
117          { /* Comp. turn */
118              /* Run the CompTurn function and get his play */
119              d = CompTurn(f,e,d);
120
121              /* Shows the question and the Comp. choice */
122              printf("How many matches do you want to take (1 to 4) : %d\n",d)
123          }
124      }
125      else
126      { /* Human versus Human */
```

A.3. The program, 21 Matches

```
127     /* Run the PlayTurn function and get the Player choice */
128     d = PlayTurn(f);
129 }
130
131 /* Take matches from the total */
132 f -= d;
133
134 /* If no one won, Go to next turn */
135 if(f > 1)
136 {
137     v++;
138     if (v > 2)
139         v = 1;
140 }
141 }
142
143 /* Show who Won */
144 printf("\nThe Winner is Player %d!\n", v);
145 }
146
147 int main()
148 { /* Main Loop, the Menu is a Loop that only stops when the User asks to Exit */
149     /* Player Option */
150     int op = 0;
151
152     /* Presentation of the Game and its Rules */
153     printf("Game of the 21 Matches\n");
154     printf("\nRules:\n");
155     printf("\tThere are 21 Matches.\n");
156     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
157     printf("\tThe Player that takes the last Match loses.\n");
158
159     /* Beginning of the Main Loop */
160     do {
161         /* Initial Menu */
162         printf("\n1) Play, Human versus Computer\n");
163         printf("2) Play, Human versus Human\n");
164         printf("0) Exit\n\n");
165
166         /* Get and Interpret the Choice */
167         printf("Choice: ");
168         scanf(" %d", &op);
169
170         switch(op)
171         {
172             case 2:
173                 /* Run the Game */
174                 Game(op);
175                 break;
176             case 1:
177                 /* Run the Game */
178                 Game(op);
179                 break;
180             case 0:
181                 /* Exit the Game */
```


A.3. The program, 21 Matches

```
182     break;
183     default:
184         /* Tell the User that he made an Invalid Choice */
185         printf("\nInvalid Number !\n");
186     }
187 } while(op != 0);
188
189 /* End of the Loop and the Game */
190 return 0;
191 }
```

Listing A.15: Source code for the 21 Matches exercise with the 7th type of plagiarism.

A.3.8 Source code for the 21 Matches exercise, with the 8th type of plagiarism.

The 8th type of plagiarism (see Section 2.3.4) is when groups of calls are turned into a function call or vice versa.

```
1 /* Library(s) used by the Program */
2 #include <stdio.h>
3
4 int CalcPlay(int r, int f, int d)
5 {
6     int x = r;
7
8     /* If all the matches are left, the Computer takes 4 matches */
9     if (f == 21)
10        x = 4;
11     else
12     {
13         if (f > 11)
14             x = 5-d + ((f-1) % 5);
15         else
16             x = 5-d;
17     }
18
19     return x;
20 }
21
22 int CompTurn(int f, char e, int d)
23 { /* Function that handles a Computer turn and returns his choice */
24     /* Saves the number of matches that the Comp. will take */
25     int r = 1;
26
27     /* If its the Comp. turn, he uses the following algorithm */
28     if (e == 'n')
29     {
30         r = CalcPlay(r, f, d);
31     }
32 }
```

A.3. The program, 21 Matches

```
33  /* Ensures that the Computer never takes more than 4 matches */
34  if (r > 4)
35      r = 4;
36
37  /* If it is not the First Player, use the following algorithm */
38  if (e == 'y')
39      r = 5-d;
40
41  /* If there are only 5 matches or less, the Computer makes the smart choice */
42  if ((f <= 5) && (f>1))
43      r = f-1;
44
45  /* Return the number of matches that the Comp. takes */
46  return r;
47 }
48
49 int AskMatches(int f)
50 {
51     /* Player Play */
52     int j = 1;
53
54     /* Is a valid Play (0 = No) */
55     int ok = 0;
56
57     do {
58         /* Asks how many Matches the Player wants to take ? */
59         printf("How many Matches will you take (1 to 4) : ");
60
61         /* Gets the number of Matches that the Player wants to take */
62         scanf(" %d",&j);
63
64         /* Verifies if the value is valid */
65         if(j < 1)
66             printf("\nNumber too Low!\n\n");
67         else if(j > 4)
68             printf("\nNumber too High!\n\n");
69
70         else if((f < 4) && (j > f))
71             printf("\nThere aren't that many matches left!\n\n");
72         else
73             ok = 1;
74     } while(ok == 0);
75
76     return j;
77 }
78
79 int PlayTurn(int f)
80 { /* Function that handles a Player turn and returns his choice */
81     /* Returns the number of Matches that the Player takes */
82     return AskMatches(f);
83 }
84
85 char AskFirst() {
86     char e = ' ';
87
```

A.3. The program, 21 Matches

```
88  /* Ask and get the choice of the Player */
89  printf("\nDo you want to be the First Player (y or n) :");
90  scanf(" %c",&e);
91
92  /* Warn the Player if the choice is invalid */
93  if ((e != 'y') && (e != 'n'))
94      printf ("\nInvalid Choice !\n");
95
96  return e;
97  }
98
99  void Game(int Type)
100 { /* Game Loop, ends when there are no matches left */
101     /* Says how many matches are left */
102     int f = 21;
103
104     /* Says if Game is versus Player or Computer */
105     char e = ' ';
106
107     /* Says whose turn it is (Player number) */
108     int v = 1;
109
110     /* Number of matches taken */
111     int d = 0;
112
113     /* On games versus Computer, ask who starts */
114     if(Type == 1)
115     {
116         do {
117             e = AskFirst();
118         } while ((e != 'y') && (e != 'n'));
119     }
120
121     /* Main Game Loop */
122     while(f > 1)
123     {
124         /* Show matches left */
125         printf("\nMatches : %d\n",f);
126
127         /* Say whose turn it is */
128         printf("Player Turn : %d\n",v);
129
130         /* Who should play, according to Game type */
131         if(Type == 1)
132         { /* Hum. versus Comp. */
133             /* See who should play */
134             if (((e == 'n') && (v == 1)) || ((e == 'y') && (v != 1)))
135             { /* Comp. turn */
136                 /* Run the CompTurn function and get his play */
137                 d = CompTurn(f,e,d);
138
139                 /* Shows the question and the Comp. choice */
140                 printf("How many matches do you want to take (1 to 4) : %d\n",d);
141             }
142             else
```

A.3. The program, 21 Matches

```
143     { /* Its the Hum. turn */
144         /* Run the PlayTurn function and get the Play. choice */
145         d = PlayTurn(f);
146     }
147 }
148 else
149 { /* Human versus Human */
150     /* Run the PlayTurn function and get the Player choice */
151     d = PlayTurn(f);
152 }
153
154 /* Take matches from the total */
155 f -= d;
156
157 /* If no one won, Go to next turn */
158 if(f > 1)
159 {
160     v++;
161     if (v > 2)
162         v = 1;
163 }
164 }
165
166 /* Show who Won */
167 printf("\nThe Winner is Player %d!\n", v);
168 }
169
170 void PrintMenu() {
171     /* Initial Menu */
172     printf("\n1) Play, Human versus Computer\n");
173     printf("2) Play, Human versus Human\n");
174     printf("0) Exit\n\n");
175 }
176
177 void PrintRules() {
178     /* Presentation of the Game and its Rules */
179     printf("Game of the 21 Matches\n");
180     printf("\nRules:\n");
181     printf("\tThere are 21 Matches.\n");
182     printf("\tEach Player can take 1 to 4 Matches in his turn.\n");
183     printf("\tThe Player that takes the last Match loses.\n");
184 }
185
186 int main()
187 { /* Main Loop, the Menu is a Loop that only stops when the User asks to Exit */
188     /* Player Option */
189     int op = 0;
190
191     /* Presentation of the Game and its Rules */
192     PrintRules();
193
194     /* Beginning of the Main Loop */
195     do {
196         /* Initial Menu */
197         PrintMenu();
```

A.3. The program, 21 Matches

```
198
199     /* Get and Interpret the Choice */
200     printf("Choice: ");
201     scanf(" %d",&op);
202
203     switch(op)
204     {
205     case 0:
206         /* Exit the Game */
207         break;
208     case 1:
209         /* Run the Game */
210         Game(op);
211         break;
212     case 2:
213         /* Run the Game */
214         Game(op);
215         break;
216     default:
217         /* Tell the User that he made an Invalid Choice */
218         printf("\nInvalid Number !\n");
219     }
220 } while(op != 0);
221
222 /* End of the Loop and the Game */
223 return 0;
224 }
```

Listing A.16: Source code for the 21 Matches exercise with the 8th type of plagiarism.

B

TEST RESULTS

This chapter contains the results obtained from using the tools with each of the source codes (see Appendix A), in the languages that are supported by the tool.

The following measures were taken to get more results, regardless of how low the match was:

JPLAG "minimal similarity of matches" was set to 1% while the "minimum match length" was left at the default value (12 for C files and 8 for Java files); item[Marble]Placed the files in a "exercise/language/year/grader/group.#/L#.extension" format, with # being the plagiarism type (from 0 to 8).

MOSS The "-m" parameter (maximum number of times a given passage may appear before it is ignored) was set to 10000; item[W. Sherlock]Got the results from the "Examine stored matches" window for the "no comments & normalized" pre-processing.

B.1 CODEMATCH RESULTS

The CodeMatch tool produces a database file (.cdb) with the results that can be exported into an HTML file.

B.1.1 *Results for the Calculator source codes*

CodeMatch returned good results (see Table 2) for the Calculator, Java source codes (see Section A.2). Note that the files were compared with themselves since CodeMatch compares the files in two folders and the same folder was used. This is a moot point as those cases got 100% matches.

B.2. JPlag results

	0	1	2	3	4	5	6	7	8
0	100	100	94	71	95	97	83	87	79
1	100	100	94	71	95	97	83	87	79
2	94	94	100	62	89	91	76	80	71
3	71	71	62	100	70	71	70	69	67
4	95	95	89	70	100	92	81	85	76
5	97	97	91	71	92	100	81	83	78
6	83	83	76	70	81	81	100	78	73
7	87	87	80	69	85	83	78	100	76
8	79	79	71	67	76	78	73	76	100

Table 2.: The results from the CodeMatch tool for the Calculator source codes

B.1.2 Results for the 21 Matches source codes

The results (see Table 3) for the 21 Matches source code (see Appendix Appendix A.3) look similar to the results for the Calculator source codes but are in fact lower (DM=-29.778). The biggest factor for this decrease was the 3rd type of plagiarism (Identifiers changed). This is probably due to the fact that the 21 Matches source codes have more identifiers than the Calculator ones.

	0	1	2	3	4	5	6	7	8
0	100	100	93	59	89	93	79	86	87
1	100	100	93	59	89	93	79	86	87
2	93	93	100	46	81	85	69	77	78
3	59	59	46	100	57	59	56	57	56
4	89	89	81	57	100	82	79	84	85
5	93	93	85	59	82	100	75	80	80
6	79	79	69	56	79	75	100	77	77
7	86	86	77	57	84	80	77	100	85
8	87	87	78	56	85	80	77	85	100

Table 3.: The results from the CodeMatch tool for the 21 Matches source codes

B.2 JPLAG RESULTS

The JPlag tool gives us results organized by the average and maximum similarities, without repeated values (see Table 4). It produces an HTML file presenting the result and allows the user to view the

B.2. JPlag results

detailed comparison of what code is suspected to be plagiarism.

B.2.1 Results for the Calculator source codes

Note that, despite our efforts, some of the results (the ones missing from the table) were not reported, indicating that there is hardly anything matching between those files. Despite the blanks, the results (see Table 4) were good since there were a lot of exact matches (100%), showing that JPlag was impervious to several types of plagiarism.

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	94.2	100	71.7	71.4	21.4
1	100	—	100	100	94.2	100	71.7	71.4	21.4
2	100	100	—	100	94.2	100	71.7	71.4	21.4
3	100	100	100	—	94.2	100	71.7	71.4	21.4
4	94.2	94.2	94.2	94.2	—	94.2	66.2	57.1	11.9
5	100	100	100	100	94.2	—	71.7	71.4	21.4
6	71.7	71.7	71.7	71.7	66.2	71.7	—	55.1	9.2
7	71.4	71.4	71.4	71.4	57.1	71.4	55.1	—	10.7
8	21.4	21.4	21.4	21.4	11.9	21.4	9.2	10.7	—

Table 4.: The results from the JPlag tool for the Calculator source codes

B.2.2 Results for the 21 Matches source codes

JPlag was better (DM=39.822) at detecting the plagiarism for this set of source codes since the results (see Table 5) are similar in relation to the exact matches and got better matches on the harder cases like the 6th, 7th and 8th types of plagiarism.

B.3. Marble results

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	90.1	100	51.9	66.9	73.8
1	100	—	100	100	90.1	100	51.9	66.9	73.8
2	100	100	—	100	90.1	100	51.9	66.9	73.8
3	100	100	100	—	90.1	100	51.9	66.9	73.8
4	90.1	90.1	90.1	90.1	—	90.1	51.9	66.0	53.9
5	100	100	100	100	90.1	—	51.9	66.9	73.8
6	51.9	51.9	51.9	51.9	51.9	51.9	—	25.1	16.2
7	66.9	66.9	66.9	66.9	66.0	66.9	25.1	—	56.3
8	73.8	73.8	73.8	73.8	53.9	73.8	16.2	56.3	—

Table 5.: The results from the JPlag tool for the 21 Matches source codes

B.3 MARBLE RESULTS

The results are presented through a suspects.nf file which has several lines in the following structure: "echo *M1 S1 S2 M2 U/S* && edit *File 1* && edit *File 2*". The *M1* and *M2* values indicate the match percentages, *S1* and *S2* give us the size of the matches and the *U/S* flag indicates if the largest percentage was found before (U) or after (S) ordering the methods.

Note that only the Calculator results are available since Marble does not support the C language.

B.3.1 Results for the Calculator source codes

The results (see Table 6) from this tool give an asymmetrical table. As expected, a few source codes accuse the movement of methods (have an S flag). This is verified in the case of the 5th, 6th and 7th types of plagiarism as they had operations, variables and statements moved, respectively.

	0	1	2	3	4	5	6	7	8
0	—	100 U	100 S	100 S	95 U	97 U	87 S	85 S	66 U
1	100 U	—	100 U	100 S	95 U	97 U	87 S	85 S	66 U
2	99 S	100 U	—	100 U	95 U	97 U	87 S	85 S	66 U
3	99 S	99 S	100 U	—	96 U	97 U	87 S	85 S	66 U
4	89 U	89 U	89 U	89 U	—	93 U	79 U	78 U	62 U
5	97 U	97 U	97 U	97 U	87 U	—	85 U	84 S	66 U
6	86 S	86 S	86 S	86 S	79 U	85 U	—	75 U	61 U
7	84 S	84 S	84 S	84 S	77 U	83 S	75 U	—	60 U
8	55 U	55 U	55 U	55 U	56 U	54 U	50 U	48 U	—

Table 6.: The results from the Marble tool for the Calculator source codes

B.4. MOSS results

B.4 MOSS RESULTS

The MOSS tool gives us the number of lines matching between each pair of files and calculates the match percentages by dividing it by the number of lines in the file though it ignores code that it finds repeatedly, thus reducing the percentages. As indicated (see Section 2.2.5), the detailed results are similar to JPlags and show the lines that match between the files.

B.4.1 Results for the Calculator source codes

As we can see in Table 7, the results are reasonable but the strategies used to avoid false positives decreased the matches (ex.: 100% to 99%). We can also notice that some types of plagiarism, namely the 6th, 7th and 8th had low matches due to their high complexity.

	0	1	2	3	4	5	6	7	8
0	—	99	99	99	83	90	51	67	57
1	99	—	99	99	83	90	51	67	57
2	99	99	—	99	83	90	51	67	57
3	99	99	99	—	83	90	51	67	57
4	82	82	82	82	—	73	50	66	39
5	90	90	90	90	74	—	39	54	47
6	48	48	48	48	48	36	—	43	10
7	67	67	67	67	67	54	46	—	22
8	44	44	44	44	31	37	12	17	—

Table 7.: The results from the MOSS tool for the Calculator source codes

B.4.2 Results for the 21 Matches source codes

As expected the increase in the size of the source codes translated into the variance of the results (see Table 8) but, since the increases were far less than the decreases it got a large negative metric value (DM=-66.889). We believe that the blank spaces indicate that the matches are very close to 0% so we considered them as 0 on the calculations.

B.5. SIM results

	0	1	2	3	4	5	6	7	8
0	—	99	99	99	90	35	62	40	79
1	99	—	99	99	90	35	62	40	79
2	99	99	—	99	90	35	62	40	79
3	99	99	99	—	90	35	62	40	79
4	89	89	89	89	—	25	61	40	60
5	35	35	35	35	25	—	7	11	16
6	61	61	61	61	61	7	—	27	37
7	40	40	40	40	40	12	27	—	15
8	68	68	68	68	52	14	33	13	—

Table 8.: The results from the MOSS tool for the 21 Matches source codes

B.5 SIM RESULTS

SIM is an interesting tool, despite not having a GUI since its results are quite detailed. The `-P` argument can be used to report the results in a `"File 1 consists for Match% of File 2 material"` format giving a quick rundown of the results. It has a few arguments that change its behavior. Unlike Sherlock however its default values seem to work well for most cases of plagiarism.

B.5.1 Results for the Calculator source codes

The results for this tool are similar to the ones from MOSS in that the results are calculated in consideration of the file sizes. This gives us an asymmetrical table. We can notice a few odd details in the results (see Table 9), namely the fact that a few files match the entirety of other files but not the other way around. An example for this is any of the 100% matches for the 8th type of plagiarism (Group of calls turned into a function call or vice versa) which are matched back by 89%. This shows how SIM is great at identifying source code that is taken from other files without missing the fact that the copy does not account for the whole thing.

B.6. Sydney's Sherlock results

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	99	99	99	99	99
1	100	—	100	100	99	99	99	99	99
2	100	100	—	100	99	99	99	99	99
3	100	100	100	—	99	99	99	99	99
4	98	98	98	98	—	97	97	98	97
5	100	100	100	100	99	—	99	99	99
6	99	99	99	99	99	99	—	99	99
7	99	99	99	99	99	99	98	—	99
8	91	91	91	91	91	91	91	91	—

Table 9.: The results from the SIM tool for the Calculator source codes

B.5.2 Results for the 21 Matches source codes

On Table 10, we can see that SIM was able to detect all types of plagiarism, given bigger source codes (DM=5.444). While we do not know which factors led to such great results, we believe that SIM is suited for larger projects where there is plenty of information accessible.

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	99	100	94	99	100
1	100	—	100	100	99	100	94	99	100
2	100	100	—	100	99	100	94	99	100
3	100	100	100	—	99	100	94	99	100
4	99	99	99	99	—	99	94	99	99
5	100	100	100	100	99	—	94	99	100
6	98	98	98	98	98	98	—	98	99
7	100	100	100	100	99	100	94	—	100
8	100	100	100	100	98	100	94	100	—

Table 10.: The results from the SIM tool for the 21 Matches source codes

B.6 SYDNEY'S SHERLOCK RESULTS

The Sherlock tool is an interesting case as its results (see Table 11) are a list of sentences in a "File 1 and File 2: Match%" format. This format does not help the user find the highest matches, it just makes the results easy to post-process. Some of the results are quite poor as the default settings are meant for fast computations. As we know (see Section 2.2.8), those settings can be tweaked in order to improve the results and must be considered alongside the results with the default settings. The

B.6. Sydney's Sherlock results

results presented are all the combinations of the settings from -n 1 to 4 and -z 0 to 5 but we will only show a few as examples.

B.6.1 Results for the Calculator source codes

With the default settings (equivalent to -n 3 -z 4), we can see that some of the results (see Table 11) were good between the 1st (Unaltered copy), 4th (Scope changed) and 7th (Statements order switched) types of plagiarism. Seeing as Sherlock is a tool for matching text documents, it makes sense that it is mostly unaffected by the movement of code but will not handle changes very well.

	0	1	2	3	4	5	6	7	8
0	—	100	31	52	100	70	64	100	63
1	100	—	31	52	100	70	64	100	63
2	31	31	—	8	31	15	10	31	20
3	52	52	8	—	52	50	52	52	40
4	100	100	31	52	—	70	64	100	63
5	70	70	15	50	70	—	61	70	45
6	64	64	10	52	64	61	—	64	47
7	100	100	31	52	100	70	64	—	63
8	63	63	20	40	63	45	47	63	—

Table 11.: The results from the Sherlock tool for the Calculator source codes

To demonstrate the effect of tweaking the -n and -z parameters, two more tables (see Tables 12 and 13) are presented each with the results of Sherlock when the arguments "-n 2" and "-z 3" were applied to it, respectively.

	0	1	2	3	4	5	6	7	8
0	—	100	57	55	94	94	77	100	76
1	100	—	57	55	94	94	77	100	76
2	57	57	—	23	55	55	42	57	40
3	55	55	23	—	52	52	55	55	52
4	94	94	55	52	—	88	73	94	72
5	94	94	55	52	88	—	73	94	72
6	77	77	42	55	73	73	—	77	66
7	100	100	57	55	94	94	77	—	76
8	76	76	40	52	72	72	66	76	—

Table 12.: The results produced by Sherlock tool for the Calculator source codes with the -n 2 argument

B.6. Sydney's Sherlock results

Table 12 shows us that using the -n parameter can greatly improve the results (DM = 106.889). The parameter changed the number of words per digital signatures, Meaning that with a smaller value, Sherlock will find plagiarism in smaller sections of source code. This translates into better matches in small changes and source code movements, but worse matches on longer modifications (as seen for the 2nd type of plagiarism).

	0	1	2	3	4	5	6	7	8
0	—	100	57	55	94	94	77	100	76
1	100	—	57	55	94	94	77	100	76
2	57	57	—	23	55	55	42	57	40
3	55	55	23	—	52	52	55	55	52
4	94	94	55	52	—	88	73	94	72
5	94	94	55	52	88	—	73	94	72
6	77	77	42	55	73	73	—	77	66
7	100	100	57	55	94	94	77	—	76
8	76	76	40	52	72	72	66	76	—

Table 13.: The results produced by Sherlock tool for the Calculator source codes with the -z 3 argument

The -z argument changes Sherlock's "granularity", a parameter that serves to discard part of the hash values. By having a value of 3, the results will be more sensitive to any changes, which resulted in small negative metric value (DM = -9.333).

Overall, we can see that Sherlock is a very specific tool and that further studies would have to be made in order to ascertain the adequate parameters to use for specific situations.

B.6.2 Results for the 21 Matches source codes

For the 21 Matches source codes, the results (see Table 14) seem more accurate (DM = 63.556) than the results for the Calculator source codes (see Table 11). This was likely due to the increased size of the source codes which made the similarities outweigh the changes.

B.7. Warwick's Sherlock results

	0	1	2	3	4	5	6	7	8
0	—	100	38	63	93	66	85	95	84
1	100	—	38	63	93	66	85	95	84
2	38	38	—	21	35	22	29	35	30
3	63	63	21	—	62	52	63	60	56
4	93	93	35	62	—	62	83	89	83
5	66	66	22	52	62	—	61	63	59
6	85	85	29	63	83	61	—	81	71
7	95	95	35	60	89	63	59	—	80
8	84	84	30	56	83	59	71	80	—

Table 14.: The results from the Sherlock tool for the 21 Matches source codes

B.7 WARWICK'S SHERLOCK RESULTS

The Sherlock tool from the University of Warwick is quite different from Sydney's. It has a GUI and allows for the selection of several options such as: The types of source code transformations used in the detection (such as Normalized and Tokenized) as well as several thresholds. It also shows results in text formats, statistics and a graph of matches (as detailed in (Joy and Luck, 1999)). The free-text results present a summary with each file and a measure that is calculated from the internal measures. Note that these show a 90% to 100% match for most of the pairs except for the L2 files (comments changed) which had 5% to 20%. Since the "no comments" transformation was selected, we believe that this decrease in the measure was because some of the comments were cut.

The following results will have the inner measures for the "no comments, normalized" and the "tokenized" source code transformations, which gives us a good idea of what is detected with each transformation behind the free-text measures.

B.7.1 Results for the Calculator source codes

Table 15 shows us what the "No Comments + Normalized" transformation on the source codes was able to detect. While these measures are not optimal, we must remember that they are internal and lead to some promising measures except for the second case (L2).

B.7. Warwick's Sherlock results

	0	1	2	3	4	5	6	7	8
0	—	98	98	56	93	82	63	88	61
1	98	—	98	56	93	82	63	88	61
2	98	98	—	56	93	82	63	88	61
3	56	56	56	—	56	56	57	41	35
4	93	93	93	56	—	77	62	83	58
5	82	82	82	56	77	—	63	70	39
6	63	63	63	57	62	63	—	59	37
7	88	88	88	41	83	70	59	—	49
8	61	61	61	35	58	39	37	49	—

Table 15.: The results from Warwick's Sherlock tool for the Calculator source codes, using the No Comments + Normalized transformation

With the Tokenized transformation, the results (see Table 16) suffer a big change which is both good (for the cases involving L0 to L5) and bad (for the results which are either lower or missing). We can see that it provides no benefit over the previous results, especially with missing values (DM = -119.333).

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	100	95	19		42
1	100	—	100	100	100	95	19		42
2	100	100	—	100	100	95	19		42
3	100	100	100	—	100	95			28
4	100	100	100	100	—	91	19		28
5	95	95	95	95	91	—			25
6	19	19	19		19		—		48
7								—	
8	42	42	42	28	28	25	48		—

Table 16.: The results from the Warwick's Sherlock tool for the Calculator source codes, using the Tokenized transformation

We can clearly see that this transformation was great for detecting plagiarism in the first 5 types of plagiarism but did a lot worse on the ones after it. This is especially true for the seventh type (statement order changed) that did not have any results presented.

B.7. Warwick's Sherlock results

B.7.2 Results for the 21 Matches source codes

For the 21 Matches source codes we can see that the No Comments + Normalized results (see Table 17) are a bit lower (DM = -25.556) but the Tokenized results are much better. This is likely due to the source codes being bigger.

	0	1	2	3	4	5	6	7	8
0	—	100	100	31	91	72	67	85	72
1	100	—	100	31	91	72	67	84	78
2	100	100	—	31	91	72	67	85	72
3	31	31	31	—	31	30	17	29	22
4	91	91	91	31	—	64	40	55	69
5	72	72	72	30	64	—	40	55	46
6	67	67	67	17	40	40	—	57	45
7	85	84	85	29	55	55	57	—	65
8	72	78	72	22	69	46	45	65	—

Table 17.: The results from Warwick's Sherlock tool for the 21 Matches source codes, using the No Comments + Normalized transformation

As we can see on Table 18, the Tokenized transformation got a lot more results and actually went beyond 100% on several cases. These results are overall better than the previous (DM = 109.556).

	0	1	2	3	4	5	6	7	8
0	—	121	121	121	72	99	82	101	72
1	121	—	121	121	67	99	82	96	73
2	121	121	—	121	72	99	82	119	72
3	121	121	121	—	59	99	82	96	73
4	72	67	72	59	—	40	52	67	86
5	99	99	99	99	40	—		56	
6	82	82	82	82	52		—	54	32
7	101	96	119	96	67	56	54	—	23
8	72	73	72	73	86		32	23	—

Table 18.: The results from the Warwick's Sherlock tool for the 21 Matches source codes, using the Tokenized transformation

We could say that similar to Sydney's Sherlock, Warwick's is also better at detecting similarities in bigger source codes. But, even though it still has some rough cases, since these results are internal and most final results have measures from 90 to 100% it is still a great improvement to the other Sherlock.

B.8. Spector results

B.8 SPECTOR RESULTS

This tool can inspect files/directories and has an option (`-output c`) to print the result(s) in the console as: *"File 1 - File 2 : calculation = resulting similarity measure"* per line.

Note that only the Calculator results are available since this tool does not support the C language.

B.8.1 Results for the Calculator source codes

The results (see Table 19) from this tool give an asymmetrical table. We can see that several cases had great similarity (0 to 4 and 7) measures and that the remaining ones were close to the results from the Code Match tool (in Table 2).

	0	1	2	3	4	5	6	7	8
0	—	100	100	100	100	94.928	77.363	100	82.494
1	100	—	100	100	100	94.928	77.363	100	82.494
2	100	100	—	100	100	94.928	77.363	100	82.494
3	100	100	100	—	100	89.573	80.142	100	77.119
4	100	100	100	100	—	94.928	77.363	100	82.494
5	95.386	95.386	95.386	90.02	95.386	—	76.578	95.386	79.643
6	70.022	70.022	70.022	69.801	70.022	68.808	—	70.022	59.954
7	100	100	100	100	100	94.928	77.363	—	82.494
8	80.86	80.86	80.86	75.519	80.86	80.171	64.002	80.86	—

Table 19.: The results from the Spector tool for the Calculator source codes

These results are mostly good indications of suspicious similarity between the files albeit some (below 70) are not strong indications.