Static Source Code Analysis Tools and their Application to the Detection of Plagiarism in Java Programs

James Hamilton
Supervised by Sebastian Danicic

June 13, 2008

Abstract

This project develops a system for detecting plagiarism in sets of student assignments written in Java. Plagiarism is viewed as a form of code obfuscation where students deliberately perform semantics preserving transformations of an original working version to pass it off as their own. In order to detect such obfuscations we assume we have a set of programs in which we attempt to find transformations that have been applied. We investigate tools for static analysis and transformation of Java programs to build a system for plagiarism detection.

Contents

Listings

List of Figures

List of Algorithms

Chapter 1

Introduction

1 Plagiarism

Plagiarism, described in the Oxford English Dictionary as "action or practice of taking someone else's work, idea, etc., and passing it off as one's own; literary theft", is prevalent in many higher education institutions worldwide and is threatening the value of degrees [1]. Many students can cheat due to the formulaic nature of degree programs and the ease of sharing information via the Internet [2]. A recent survey [3] has also shown that many universities are reluctant to punish acts of plagiarism and that plagiarism is more prevalent in postgraduate courses. Plagiarism also occurs commercially for example in the high-profile SCO vs IBM case [88].

In Computer Science and Software Engineering subjects students must write their own programs in order to gain an understanding of programming. It is very easy for students to work together or copy other students source code and modify the code to pass it off as their own.

In 2000, Joint Information Systems Committee (JISC), an advisory committee for UK higher-eduction ICT, commissioned a survey into source-code plagiarism in UK higher education institutions [41]. The report provides an analysis of plagiarism in UK higher education institutions, analysis of two plagiarism detection systems, and recommendations for JISC. The report concluded that there is a definite problem of source code plagiarism in UK higher education institutions, and recommends the use of the systems presented for institutions to detect such plagiarism. The study also found that the two systems, MOSS[1] and JPlag[2], gave widely different results. MOSS is described as using a document fingerprinting technique [86] and JPlag "does not merely compare bytes of text, but is aware of programming language syntax and program structure"[81]. The complete workings of both systems are kept a secret in order to prevent analysis and circumvention [41]. Both systems are online services provided to registered users (free registration).

We are concerned with plagiarism detection of sets of source code, for example student answers for a programming assignment i.e. given a set of source files which are most similar?

Clough[20] reviewed several systems to aid in the detection of plagiarism in both natural and programming languages and found that structural based methods were often a better approach than attribute counting methods and, as a closing thought, notes that human intervention will always be needed to provide the final verdict. Another promising alternative is AC[3] which is open-source and provides an interesting visualisation of its results.

1.1 Similarity

Detecting plagiarism in software presents some interesting problems due to the nature of programming. Defining similarity is not necessarily a trivial task, especially in terms of software source code, as many attributes of a set of non-plagiarised programs could exhibit similar properties. The reasons for program similarity can be summarised in six categories (Figure 1.1) only one of which is plagiarism [105]. Walenstein *et al.* discuss types of similarity and measures [102] . Textual, syntatic and structural similarities are briefly discussed though not in depth and some suggestions are made for measures of similarities including textual, metric, features, shared information, program execution, input-output, semantic distance and program dependency graph similarity.

[1]http://theory.stanford.edu/ aiken/moss/
[2]https://www.ipd.uni-karlsruhe.de/jplag/
[3]http://tangow.ii.uam.es/ac/

Third Party Source Code from, for example open-source or purchases libraries, could be used in more than one program.

Code Generation Tools such as Netbeans IDE produce code for the user, for example code for a Graphical User Interface (GUI).

Commonly Used Identifier Names such as *result* or *i* existing due to prevalence of their use in teaching.

Common Algorithms would be implemented in the same way.

Common Authors of more than one program result in code written in a certain style.

Copied Code either authorised or plagiarised

Figure 1.1: Zeidman lists 6 categories of software similarities [105]

Software plagiarism detection in academia introduces an additional problem: many software assignments are standardised and formulaic, especially in beginner programming courses [13]. For example tasks such as 'Write a program which emulates rolling a die' or 'Write a program to calculate leap years' [27] will most likely result in very similar programs from all students even if they work independently. Although not an easy problem to over-come it may be possible to analyse the programming style used to determine who authored a program [57].

Types of Plagiarism

Plagiarism of program source code can be achieved by students manually applying transformations to another students source code. These transformations could be very simple such as adding, removing or changing comments in a program and renaming variable names or the transformation could be more complicated such as structure changing transformations such as in-lining methods and changing for-loops to while-loops. Whale lists many types of plagiarism that students could engaged in [103] ranging from simple to complex. Faidhi and Robinson defined 6 levels of source code plagiarism (Figure 1.2) from simple to more complicated methods [39].

Level 1 comments e.g. add, remove or change comments

Level 2 identifiers e.g. rename identifiers

Level 3 code positions e.g. move field variable declarations from the top of the source to the bottom

Level 4 procedure combination e.g. in-lining procedures

Level 5 program statements e.g. rearranging program statements

Level 6 control logic e.g. changing for-loops to while-loops

Figure 1.2: Faidhi and Robinson defined 6 levels of source code plagiarism [39]

1.2 Examples of Plagiarism

Identifier Names

A common method, falling into level 2, which students use to plagiarise is modifying identifier names, for example changing a variable named *total* to *i*. For example the programs 1.1, 1.2 and 1.3 are all identical apart from the identifier names. Students who copied the original program could just have changed the names of identifiers. The program would still produce the same output. This is not a very good plagiarism, as it is obvious from the examples that they are similar. In these small examples it is obvious to a marker by just looking at them that they are similar. The main method cannot be renamed.

A simple technique of tokenising the programs and replacing the variables with markers, ignoring the actual names, would result in matches between all programs.

Listing 1.1: Variable Change Program A

```
public class VarChangeA {

        public static void main(String[] args) {
                for(int i = 1; i <= 10; i++)
                        System.out.println(i + ": " + factorial(i));
        }

        public static int factorial(int n) {
                if(n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }
}
```

Listing 1.2: Variable Change Program B

```
public class VarChangeB {

        public static void main(String[] n) {
                for(int b = 1; b <= 10; b++)
                        System.out.println(b + ": " + f(b));
        }

        public static int f(int a) {
                if(a <= 1)
                        return 1;
                else
                        return a * f(a - 1);
        }
}
```

Listing 1.3: Variable Change Program C

```
public class VarChangeC {

        public static void main(String[] arguments) {
                for(int variableA = 1; variableA <= 10; variableA++)
                        System.out.println(variableA + ": "
                                                + functionA(variableA));
        }

        public static int functionA(int variableB) {
                if(variableB <= 1)
                        return 1;
                else
                        return variableB * functionA(variableB - 1);
        }
}
```

Comments

Program 1.4 is completely identical to program 1.1 except that comments have been added. The two programs are obviously plagiarised and the comments do not make a difference to the actual code. Simply ignoring comments when comparing programs will match the two programs.

Program Restructuring

A simple program restructuring would be moving class field declarations from the top of the class in the original source to the bottom in the plagiarised source. Program 1.5 is a simple class containing 3 field definitions and some method definitions. Program 1.6 is a plagiarised version of 1.5 where the definitions have been moved around slightly.

Another example of restructuring involves swapping *if* and *else* statements by negating the original *if* condition so that, for example, program block 1.7 becomes program block 1.8.

Inlining

Procedures can be combined and reduced into one single procedure. Two methods in 1.5 have, in program 1.9, been combined into the constructor. The two programs are semantically identical but 1.9 has fewer methods.

Loops

In languages with C-like syntax, such as Java, a while-loop 1.11 is a for-loop 1.10 with no initialisation or counter statements and both can be inter-changed easily [31] (Listings 1.10 and 1.11). Program 1.12 is the same as program 1.1 but has a while-loop in place of the for-loop.

1.3 Java Specific Techniques

Java (and some other languages) has some features which could allow students to plagiarise easily without a marking realising. For example a student could take a program with generics and remove generics from it (or vice-versa). The marker may not realise at first glance that two programs are similar, and in fact the removal or addition of generics to a program may fool a plagiarism detection system based on string counting methods as it removes or adds some text to the program.

Listing 1.4: Adding Comments

```
/*  Program: A program to output numbers 1 - 10 and their factorials
        Author: James
        Date:   12/01/08
*/

public class VarChangeAWithComments {

        /* main method */
        public static void main(String[] args) {
                /* iterate through numbers 1 - 10 */
                for(int i = 1; i <= 10; i++)
                /* print out the number i and it's factorial */
                        System.out.println(i + ":␣" + factorial(i));
        }

        /* method to calculate factorial for a number */
        public static int factorial(int n) {
                /* computing factorial */
                if(n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }
}
```

Listing 1.5: A simple class with field and method definitions

```
public class RestructureA {

        private int x = 5;
        private int y = 3;
        private int z = 2;

        public static void main(String[] args) {
                new RestructureA();
        }

        public RestructureA() {
                doAnotherThing();
        }

        public void doSomething() {
                System.out.println(x);
        }

        public void doAnotherThing() {
                System.out.println(z);
                doSomething();
        }

        public String toString() {
                System.out.println("x:␣" + x + ",␣y:␣" + y + ",␣z:␣" + z);
        }
}
```

Listing 1.6: Restructured version of 1.5

```java
public class RestructureB {

        public void doAnotherThing() {
                System.out.println(z);
                doSomething();
        }

        public RestructureB() {
                doAnotherThing();
        }

        public void doSomething() {
                System.out.println(x);
        }

        public String toString() {
                System.out.println("x: " + x + ", y: " + y + ", z: " + z);
        }

        private int x = 5;
        private int y = 3;
        private int z = 2;

        public static void main(String[] args) {
                new RestructureB();
        }
}
```

Listing 1.7: Inlined version of 1.5

```java
if(true) {
        //do this
}else{
        //do that
}
```

Listing 1.8: Inlined version of 1.5

```java
if(!true) {
        //do that
}else{
        //do this

}
```

Listing 1.9: Inlined version of 1.5

```java
public class RestructureC {

        private int x = 5;
        private int y = 3;
        private int z = 2;

        public RestructureC() {
                System.out.println(z);
                System.out.println(x);
        }

        public String toString() {
                System.out.println("x: " + x + ", y: " + y + ", z: " + z);
        }

        public static void main(String[] args) {
                new RestructureC();
        }
}
```

Listing 1.10: For Loop

```java

for( initialiser; condition; counter ) {
        //statements
}
```

Listing 1.11: While Loop

```java

initialiser;

while(condition) {
        //statements

        counter;
}
```

Listing 1.12: For Loop → While Loop

```
public class ForWhileChangeA {

        public static void main(String[] args) {
                int i = 1;

                while(i <= 10) {
                        System.out.println(i + ": " + factorial(i));
                        i++;
                }
        }

        public static int factorial(int n) {
                if(n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }
}
```

Listing 1.13: For Loop → While Loop

```
public class A {

        Vector<Integer> v = new Vector<Integer>();

        public A() {
                ArrayList<A> b = new ArrayList<A>();
        }

}
```

Listing 1.14: For Loop → While Loop

```
public class B {

        public B() {
                ArrayList myArrayList = new ArrayList();
        }

        Vector myVector= new Vector();

}
```

2 Code Obfuscation

Code obfuscation is the transformation of source code in such a way that makes it unintelligible to human readers of the code and reverse engineering tools, such as program slicing tools. Obfuscation is a semantics preserving transformation applied to program source code in the same way as plagiarism.

An obfuscator should meet three conditions: functionality must be maintained, the obfuscated code must be efficient and the code must be obfuscated [8]. The third condition meaning that the obfuscated program should be sufficiently changed such that it is difficult to understand if it is reverse engineered.

The use of byte code for a distributed program, for example Java Byte Code, increases the possibility of reverse engineering the application, as byte code is at a higher level than machine code. Although it is possible to reverse engineer machine code [61] the widespread use of Java increases the need to use obfuscation to protect intellectual property contained within applications due to the relative ease of reverse engineering [85]. The protection of intellectual property is the main reason for the need of code obfuscation, a company needs to protect its intellectual property in the software it distributes to prevent rivals gaining a competitive advantage.

A generalised obfuscator for all programs is impossible to implement but it is possible that a restricted definition of obfuscation is possible [8].

Plagiarism is a type of obfuscation applied to source code by students but the aim, instead of making the program unreadable, is to create a semantically identical program with enough implementation differences to confound their assignment marker. Many obfuscation techniques are therefore useful for students to use as plagiarism techniques and detecting plagiarism may also help detecting and reverse engineering obfuscated code.

It is possible that students could use code obfuscation systems to produce plagiarised source code instead of manually applying plagiarism transformations. Collberg *et al* tested this idea with several experiments to test MOSS with submissions plagiarised automtically with the SANDMARK[4] framework tools using Java source files [22]. The experiments showed that in most cases the transformations made by the obfuscator where enough to fool the plagiarism detection system. Peculiarly, the act of compilation and decompilation[5] on it's own is enough to fool MOSS. One technique suggested to identifying the students own work is software watermarking [24] before submission to embed each student's identification number in their program [22].

Something not investigated is the implementation of an automatic plagiariser i.e. a system that takes one input source and produces many semantically identical but somehow different sources to fool a plagiarism detection system (Figure 1.3).

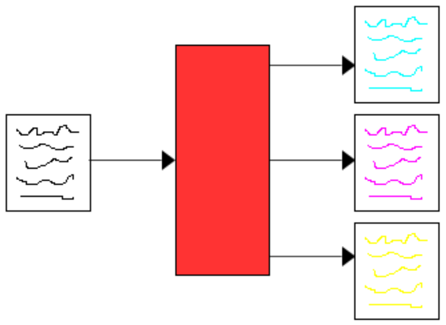


Figure 1.3: Automatic Plagiariser - one student provides an input program to produce many plagiarised versions for their classmates.

[4]http://sandmark.cs.arizona.edu/

[5]the compilation/decompilation step was actually required as SANDMARK works only on Java class files. A commercial decompiler from http://ahpah.com/ was used for this.

2.1 Examples of Code Obfuscation

There are many different techniques to obfuscate source code and Collberg *et al* provide a taxonomy of many code obfuscation transformations [23]. A selection of these will be presented here. It is easy to see the similarity between the examples listed in section 1.2 and the examples of code obfuscation techniques.

Removing Comments

In many cases comments are invaluable to understanding a program properly [72]. Comments explain the thought processes of the programmer, allowing another to understand clearly what the code is doing. For example listing 1.15 shows a simple method that would probably not be understandable to anyone other than the author without comments documenting why the calculations are happening the way they are. Compiled languages such as *Java* remove comments, so it is not a problem in distributed *Java* byte code. Removing comments is more important for a scripting language such as *JavaScript* where the full source code is visible.

Listing 1.15: Comments help to understand a program

```
//method to calculate......
public int calc(int x, int y) {
        int z = 7 * y;                    //first multiply.....
        y = x * z;                        //then do this....
        return y * 3;                     //finally return....
}
```

Identifier Name Randomisation

Well written source code is self-documenting [74] - if variable and method names are chosen wisely they clearly explain what is happening in the program. The aim of name randomisation is to remove the meaning inferred by useful names (Listing 1.16), and replace them with meaningless names (Listing 1.17).

Listing 1.16: Before identifier randomisation

```
public static final double TAX_RATE = 15;

public int calculateTax(Product product) {
        double price = product.getValue();
        return price + ( price * TAX_RATE );
}
```

Listing 1.17: After identifier randomisation

```
public static final double asfkfiasdy8213jhg23 = 15;

public int jk2h138f7s7d8f(sjdh862312 aksjdshagd2lkjLKAJMNCX) {
        double msd99227djdg = aksjdshagd2lkjLKAJMNCX.kv00238123bfsdf();
        return msd99227djdg + ( msd99227djdg * asfkfiasdy8213jhg23 );
}
```

Control Flow Obfuscation

A clear flow of a program can be useful in deducing what is happening. Introducing unnecessary code which never executes (using, for example, an *if statement*) will make understand the flow of the program more difficult.

Variable Values

Values could easily be complicated by applying a mathematical function to them. For example replacing the declaration $i = 5$ with $i = 5 * 10 / (3 * 3 + 1)$

Inline Procedure

Grouping code together in separate procedures aids understanding of a program. Replacing procedures calls could complicate a programs structure somewhat.

3 Static Analysis and Program Transformation

3.1 Programming Languages

Programmers write programs in a textual format which is then parsed by another program, known as a parser, in order to produce an in-memory abstract representation of the program in the form of an AST [4]. Transformations and Static Analysis operations can then be carried out on the AST representing the program. Static analysis operations involve analysing program code without executing it and transformation operations involve the transformation of a program in some predictable way.

In general, a language is a means of communicating by the use of sounds or symbols. In computing, an artificial language (i.e. a programming language) allows a programmer to communicate with a computer to tell the computer what to do.

Languages, natural or artificial, are defined by a set of syntax, semantic and pragmatic rules and a set of symbols which make up the alphabet.

An *alphabet*, usually denoted Σ, is simply the set of symbols used in a language. A finite sequence of symbols is a string.

The set of all strings in a language, defined by $\Sigma^\star$ is infinite due to the length of strings being unbounded.

A language represents a subset of $\Sigma^\star$ as there are usually restrictions on the strings used in a language. In the English language the set of strings must be valid words. In a computer language the set of strings is defined using a grammar.

Grammars

A grammar precisely defines a formal language, such as a computer language, that is it defines all the possible symbols and sequences of symbols which make up words for a language. It does not define the semantics of the language. A grammar is used to recognise a string of a source language, which in computer science, is known as parsing.

In the 1950's, Chomsky [18] described four classes of grammar: Recursively enumerable, Context-sensitive, Context-free and Regular. The latter two became useful for describing computer languages.

Every programming language can be described by a grammar which many tools use to create a parser to parse source code of that language. A standard syntax for writing grammars is Backus Naur Form (BNF) which is composed of *terminals* and *non-terminals*. Terminals consist of the special, reserved symbols which make up a language such as *if* or =. Non-terminals represent the syntactic rules of a language and are composed of terminals and other non-terminals. In BNF non-terminals are usually enclosed in <> symbols.

BNF is made up of production rules composed of terminals and/or non-terminals in the format $P ::= x$ where P is a non-terminal and x is a list of terminals and/or non-terminals. The right-hand-side can consist of more than one alternative by using the | symbol to separate the alternatives $P ::= x \mid y \mid z$. There exists extensions to BNF such as EBNF and others which include syntax such a regular expression operators, for example figure 1.4, page 19 defines an Identifier non-terminal as a Letter terminal followed by zero or more Number or Letter terminals.

```
<Identifier> ::= Letter ( Number | Letter )*
```

Figure 1.4: EBNF example

Parsing

Computer programs written in a programming language must be understood somehow by the computer which needs to execute the program. Lexical analysis and parsing allow a computer to take a source program, understand it and translate it to machine code for execution. Figure 1.5, page 20 shows the flow of a parser system which takes a source program as input and parsers it.

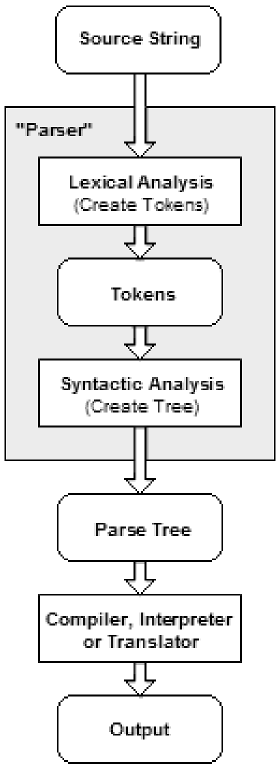


Figure 1.5: Parser Flow. Source: Wikipedia

The lexical analyser turns the source file into a stream of tokens which are passed to the syntactic analyser which creates a parse tree representing the program in-memory. The tokens that the lexical analyser generates are made up of lexemes and their token-types (Figure 1.6, page 20).

IDENTIFIER	ASSIGNMENT	NUMBER	MULTIPLY	NUMBER	ADD	NUMBER
x	=	3	*	5	+	4

Figure 1.6: Example Token Stream for x = 3 * 5 + 4

The syntactical analyser receives this token stream and creates a parse-tree - an in-memory representation of a program in the form of a tree where branches represent the production rules and leaves the symbols (Figure 1.7, page 21).

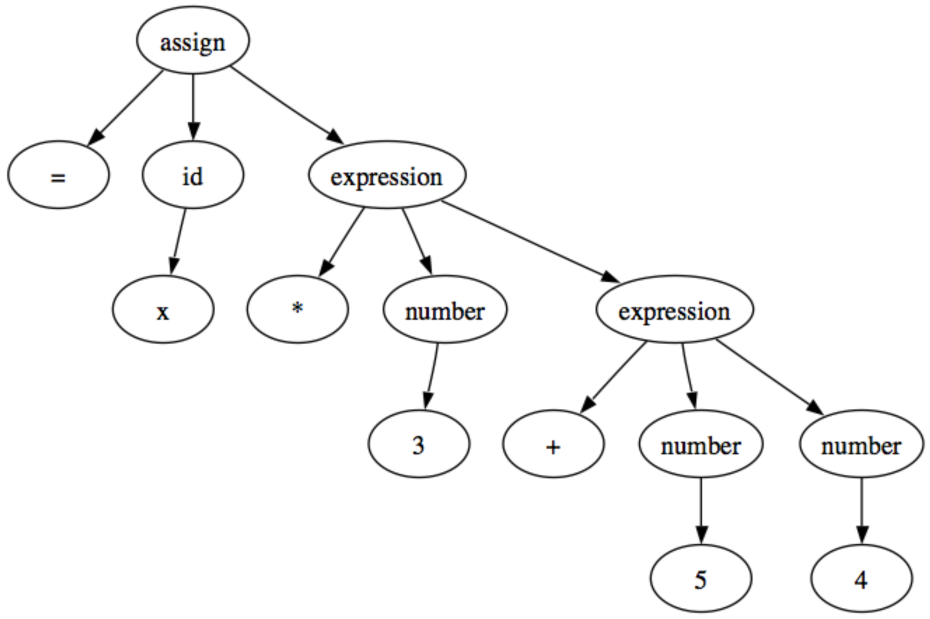


Figure 1.7: Example parse tree for x = 3 * 5 + 4

An AST is a finite, labelled, directed graph representing the structure of a program. The nodes of an AST represent the language constructs, and a node's children represent the values on which the language constructs operate. It is an abstract representation which doesn't store redundant information as a parse tree does (Figure 1.8, page 21).

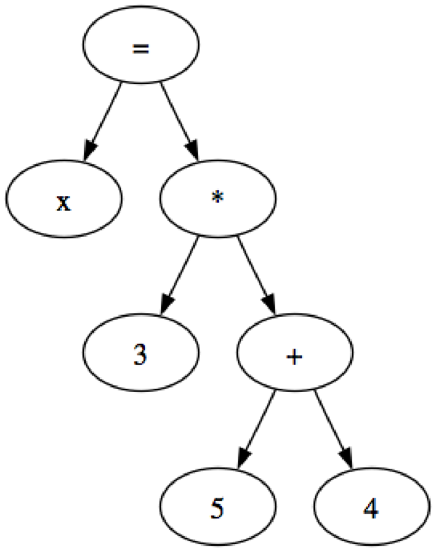


Figure 1.8: Example AST for x = 3 * 5 + 4

Java

In the last decade Java [44] has become a very popular programming language [49] in industry and education. Java's two main appeals are platform independence and automatic memory management [99]. Programmers can write code once and run it on any platform for which a Java Virtual Machine (JVM) has been created and not have to worry about managing memory like in a language like C.

Java was designed with networking in mind which has become an important feature of the language due to the growth of networked technology nowadays. It was originally destined for powering set-top boxes but its creators realised its potential and it has become one of the most popular programming languages [12].

Since it's popularity has increased many university computing departments have started teaching Java instead of the more tradition C or C++ [66], though not everyone agrees this is a good move [29, 79].

As of May 2007 Sun open-sourced most of it's Java Development Kit, including *javac* the Java Compiler [67].

From version 1.5 the language includes new features [92] such as Generics, Annotations, Auto-boxing and the new for-each construct. These features, introduced in version 1.5, are not compatible with older Java Virtual Machines and version 1.5 is described as a "major feature release" [92]. The latest version, Java 1.6[6], adds even more new features including the Compiler Application Programmers Interface (API) [94]. The new Compiler API allows programatic invocation of the java compiler, it also allows for a programmer to parse Java source code and traverse it's AST [6].

Example 1.18 shows a use of Generics in Java. The class Test holds an array of objects, of unknown type (though stored as an Object). The type is chosen by the programmer when Test is instantiated. This allows only the type of object that the programmer wants to be added to the array in the class. A compile time exception will be thrown if another type is added. Compare this to, before generics, having the check the type of each object being added or got from the class Test.

The Visitor Pattern

A common design pattern to traverse trees in Java (Listing 1.19) is by using The Visitor Pattern [51] - which is a way of separating an algorithm from an object structure allowing the modification of the algorithms without changing the object structure [43].

Each object to be visited will implement a method *accept* which accepts a *Visitor* object. The Visitor object will implement *visit* methods for each of the concrete classes to be visited. The visitor pattern simulates double-dispatch in Java which is a single-dispatch language. The double-dispatch mechanism dispatches a function call to different concrete classes depending on the type of the objects involved [63].

[6]`http://java.sun.com/javase/6/`

Listing 1.18: Java Example Using Generics

```
public class Test<P> {

        private Object[] items;
        private int c = 0;

        public Test(int size) {
                items = new Object[size];

                for(Object i : items) {
                        i = null;
                }
        }

        public void add(P p) {
                items[c++] = p;
        }

        public P get(int i) {
                return (P)items[i];
        }

        public int size() {
                return items.length;
        }

        public static void main(String[] args) {
                Test<String> t = new Test<String>(5);

                t.add("hello");

                //t.add(1); throws a compile time exception

                System.out.println(t.get(0));

                Test<Integer> n = new Test<Integer>(5);

                //n.add("hello"); throws a compile time exception

                n.add(1);

                System.out.println(n.get(0));

        }
}
```

Listing 1.19: The Visitor Pattern in Java

```java
public interface Visitor {
        public abstract void visit(AddExpression e);
        // abstract visit methods for all concrete classes
}

public class AddExpression {
        public int left;
        public int right;

        public AddExpression(int left, int right) {
                this.left = left;
                this.right = right;
        }

        public void accept(Visitor v) {
                v.visit(this);
        }
}

public MyVisitor implements Visitor {
        public void visit(AddExpression e) {
                System.out.println(e.left + e.right);
        }
}

public static void main(String[] args) {
        AddExpression e = new AddExpression(3, 3);

        MyVisitor v = new MyVisitor();

        e.accept(v);
}
```

3.2 Static Analysis

Static analysis involves analysing program code without executing it, as opposed to dynamic analysis which executes code. Static analysis tools create abstract syntax trees representing the program to be analysed from source code. We are interested in using Java 1.5 source code as input, building an AST and analysing the program in some way.

Static analysis is a means of program testing without executing the program meaning that the tests can be applied at an earlier stage in the development, and on sections of a program. Due to the size of software projects it can often be difficult to test the whole programs therefore static analysis tools can be used throughout development to analyse the on-going project for errors as they are being created [55]. This is not to say that dynamic analysis can be ignored - it too is an important part of the software testing process [36].

Static analysis is useful for analysing source code to find errors that are not immediately obvious - for example if a programmer accidently added a semi-colon in a for-loop (Listing 1.20, page 25). The program would compile but there would be no code executed as part of the for-loop.

Statically analysing code removes the need to compile a program before it can be tested and "it is an excellent tool to look for errors of a syntactical nature such as unreachable execution paths, type mismatches, and poor code constructs" [40].

Another example of static analysis is program slicing which is used to identify all code that affects a given variable in a program. This can be used to test the behaviour of variables independent of executing the whole program.

Static analysis tools are also useful to find security flaws in programs by, for example, identifying common mistakes by programmers such as using function calls in a way known to cause security problems [16].

Rice's Theorem [82] states, informally, that all interesting questions that can be asked about a program are undecidable. The theorem, which in essence can be reduced to the Halting Problem [98, 87, 16], means that we can only gain approximate answers to our non-trivial questions and therefore static analysis is undecidable [58].

Static analysis tools can be used to detect plagiarism in programs [97] and, related to this, advise students on their programming skills in order to give them feedback without a tutor [68, 34]. The tools can also be used to detect obfuscation (See page 16 for more about code obfuscation) and malicious code hidden via the use of obfuscation techniques [19].

Listing 1.20: Where static analysis might be useful in Java

```
//Java
class SA {

        public static void main(String[] args) {
                for(int i = 0; i < 10; i++);
                        System.out.println("Hello World");
        }

}
```

3.3 Program Transformation

Program transformation is the act of changing one program into another which can be divided into two categories: translation and rephrasing [101]. Translation is concerned with transforming a program from a source language into a different target language (Listing 1.21), whereas rephrasing is concerned with transformation of a program into a different program in the same language (Listing 1.22). These can be further subdivided into into 20 different sub-categories each specialising in a different area of program transformations [101].

Some transformations preserve the semantics or a program while other transformations change semantics in predictable ways. Transformation operations can be applied to a program to optimise it, and produce a more efficient program which preserves the semantics of the original [64].

Plagiarism is a form of semantics preserving transformation applied manually by a student in order to cheat, for example, in a programming assignment [105] (See page 8 for more about plagiarism).

Code obfuscation is a semantics preserving transformation applied by automatic software systems to attempt to hide the implementation of a program and make de-compiling of the program more difficult [85, 61] (See page 16 for more about code obfuscation).

Listing 1.21: Translation from Java to C#

```
//Java
class Translation {

        public static void main(String[] args) {
                System.out.println("Hello World");
        }

}

//C#
using System;

class Translation {

        public static void Main(string[] args) {
                Console.WriteLine("Hello World");
        }

}
```

Program transformations are usually applied to an AST, where some transformation is done to the AST which can then be compiled, executed or output as source code via the use of a pretty printer.

We are only concerned with rephrasing transformations of Java 1.5 programs, though some tools analysed will be of general use.

Listing 1.22: Java For to While Loop

```
class Rephrasing {

        public static void main(String[] args) {

                for(int i = 0; i < 10; i++) {

                        System.out.println("Hello World");

                }

        }

}

class Rephrasing {

        public static void main(String[] args) {

                int i = 0;

                while(i < 10) {

                        System.out.println("Hello World");

                        i++;
                }
        }

}
```

4 Contributions of this Project

This introduction has provided some background information on the subjects of plagiarism, obfuscation, static analysis and program transformation. The link between plagiarism and obfuscation has also been suggested.

Chapter 2 surveys a selection of tools available for static analysis and transformation of Java programs and suggests an appropriate tool for the implementation of a plagiarism and obfuscation detection system.

Chapter 3 surveys techniques for detecting plagiarism ranging from simple methods such as considering program metrics, to factorisation, winnowing, byte code analysis, parse tree analysis, call graph analysis.

Chapter 4 describes the implementation of a plagiarism detection system using some of the techniques from chapter 3 and some new techniques.

Chapter 5 puts the plagiarism detection system to test using some real data from a first year Java programming course[7] at Goldsmiths, University of London.

Chapter 6 discusses the results of the implementation and suggests areas for further work.

[7] http://seb.doc.gold.ac.uk/cis109/

Chapter 2

Survey of Static Analysis and Program Transformation Tools

In all the examples of plagiarism and obfuscation the struture of a program is changed somewhat by applying a semantics preserving transformation. So in order to detect plagiarism we need tools to analyse programs. This chapter takes a look at some available tools for static analysis and transformation of Java 1.5 programs.

Several open source tools for the static analysis and transformation of Java 1.5 programs will be tested to find the most effective. Many of the tools are designed for any language via the use of grammars defining them. On the other hand, some tools such as the Java Compiler are only useful in the context of Java static analysis and/or transformation. The focus will be on the Java language – specifically the latest versions 1.5 (a major release with features such as Generics, Annotations).

An analysis of the following tools will be performed, the findings will be analysed and the easiest to use will be selected. The minimum requirement for a tool to be considered is that it includes or generates a lexical analyser and parser, and can build an AST representing the program to be transformed. Another requirement is that the tool must be open-source, thus allowing ease of modification in anyway. The tools include: JavaCC, TXL, ANTLR, Sun's Java Compiler, Eclipse Platform.

The tasks of plagiarism detection and code obfuscation will be taken in to account when choosing an appropriate system. Some systems are likely to be more suited to these tasks than others. The system determined to be most appropriate for the task will be used to implement a plagiarism detector for Java 1.5 source code.

1 ANTLR

ANother Tool for Language Recognition (ANTLR) is a predicated LL(k) parser generator [77]. It is capable of generating a parser in many target languages [37] including C++, Java, Python and C# - though we are only concerned with parsers in the Java language. ANTLR relies on a grammar specification [38] of the input language in a format similar to EBNF. The generated parser can be extended for static analysis and transformation operations [76].

The Java 1.5 Grammar available on the ANTLR website[1] will be used in conjunction with ANTLR to build a Java lexer and parser. ANTLR Works[2], a GUI designed to aid development and debugging of ANTLR grammars, is also available from the ANTLR web site.

ANTLR takes as input a grammar and outputs a lexer and parser for that grammar. The outputted lexer and parser will be Java programs, and invoked from another Java program (Listing 2.7) which will parse a Java file from standard input. Actions can be added to ANTLR grammar files in the target language which will be included as part of the generated parser. When the parser is invoked these actions are also invoked in the appropriate places. Tree generation rules can also be added to an ANTLR grammar which will then build an AST as the source is parsed. Though the tree generation stage is not necessarily needed as the action can be embedded in the grammar for many operations.

1.1 ANTLR Grammar Language

The ANTLR grammar language has a similar syntax to EBNF and is briefly described here[3]. The language consists of lexical and grammatical rules. Lexical rules begin with an uppercase letter and describe characters (Listing 2.3), whereas parser rules describe tokens by using the lexical rules. An ANTLR grammar file begins with it's name, and then a list of rules (Listing 2.1).

Listing 2.1: ANTLR Grammar file header

```
grammar GrammarName; //Name must match filename

<rules>
```

Rules take the form show in Listing 2.2 where Definition is a list of one or more alternatives, separated by `|`, which match the rule. Definitions can contain actions to be taken by the lexer or parser. Such actions can contain target language code which will be executed during the parsing of each rule (here we only deal with Java), as well as referring to ANTLR functions and rules.

Listing 2.2: ANTLR General Rule Form

```
RuleName : Definition ;
```

Listing 2.3: ANTLR Lexer rule defining white space characters

```
//includes an action to set the Token Channel to hidden.
WS  :  ' '|'\r'|'\t'|'\u000C'|'\n' { $channel=HIDDEN; } ;
```

Ranges are defined using .., for example the range ‘a’..‘z’ consists of all the lowercase Latin alphabet characters and ‘A’..‘Z’ consists of all uppercase. Java supports Unicode characters, so the grammar defines sequences of such characters which are to be tokenised as letters. In ANTLR unicode characters must be escaped (Listing 2.4).

Rule definitions refer to other rules (and possibly themselves) as part of their definition. For example, in Java, an identifier is defined as a letter followed by zero or more letters or digits. Asterix

[1]Java 1.5 Grammar for ANTLRv3 written by Terrence Parr, and updated by Koen Vanderkimpen & Marko van Dooren, Chris Hogue and Hiroaki Nakamura

[2]`http://www.antlr.org/works/index.html`

[3]See ‘The Definitive ANTLR Reference: Building Domain-Specific Languages’ [76] for a complete description of ANTLR and grammar development

Listing 2.4: ANTLR Lexer Rule defining Java Letter Ranges

```
        Letter
    :   '\u0024' |
        '\u0041'..'\u005a' |
        '\u005f' |
        '\u0061'..'\u007a' |
        '\u00c0'..'\u00d6' |
        '\u00d8'..'\u00f6' |
        '\u00f8'..'\u00ff' |
        '\u0100'..'\u1fff' |
        '\u3040'..'\u318f' |
        '\u3300'..'\u337f' |
        '\u3400'..'\u3d2d' |
        '\u4e00'..'\u9fff' |
        '\uf900'..'\ufaff'
    ;
```

(*) is used to denote 'zero or more', while plus (+) denotes 'one or more' and question mark (?) denotes 'none or one' (Listing 2.5).

Listing 2.5: ANTLR Identifier definition

```
Identifier  :   Letter (Letter | JavaIDDigit)*   ;
```

The start symbol for parsing a Java file is the compilationUnit rule. A Java program is defined as 'none or one annotation followed by none or one package declaration followed by zero or more import declarations followed by zero or more type declarations' (Listing 2.6).

Listing 2.6: ANTLR Java Compilation Unit Rule

```
        compilationUnit :
                annotations?
                packageDeclaration?
                importDeclaration*
                        typeDeclaration*
        ;
```

1.2 ANTLR Invocation

Once a parser is generated from an ANTLR grammar a simple Java program (Listing 2.7) can be used to invoke it. This program can then be used as a basis for writing software to perform static analysis and program transformation tools.

Listing 2.7: Java Program to invoke ANTLR lexer and parser

```
import org.antlr.runtime.*;

public class Test {

        /*
        Invoke via command line e.g.
        java Test < Test.java
        */

        public static void main(String... args) throws Exception {

                ANTLRInputStream input = new ANTLRInputStream(System.in);

                JavaLexer lexer = new JavaLexer(input);

                CommonTokenStream tokens = new CommonTokenStream(lexer);

                JavaParser parser = new JavaParser(tokens);

                //compilationUnit is the start symbol
                parser.compilationUnit();

        }

}
```

2 TXL

TXL, a tree rewriting language which is not Java specific, "is a unique programming language specifically designed to support computer software analysis and source transformation tasks" [25] TXL requires the language specification in the form of a grammar file, similar to JavaCC. Again grammars for Java 1.5 have been created, and are freely available[4]. Use of a Java 1.5 grammar will allow transformations to be performed on Java 1.5 programs using the TXL rewriting language.

TXL is a program transformation language which is "specifically designed to support computer software analysis and source transformation tasks". It is a hybrid functional and rule-based language, divided into two parts: the source language definition in as a BNF grammar and a set of transformation rules. A grammar for Java 1.5 is available on the TXL website.

2.1 TXL Grammar Language

The language definition is given as a BNF grammar.

Non-terminals

The define statement is used to define non-terminals which can each contain a list of alternative forms of that non-terminal (Listing 2.9).

Non-terminals can be redefined later on in a grammar, to over-ride or extend their definitions, using the redefine statement (Listing 2.10). There are several predefined non-terminal types: id, number, stringlit, charlit and comment. There are also some special non-terminals: empty - which is always matched regardless of input, but consumes no tokens, any - which matches any non-terminal type.

When referenced in a non-terminal definition non-terminal names must appear in square brackets.

The start symbol is defined as *program*.

Non-terminal Modifiers

There are three important non-terminal modifiers are opt, repeat, list which can be applied to a non-terminal by including the modifier in the non-terminal reference (Listing 2.8).

Listing 2.8: TXL non-terminal modifier syntax

```
[modifier non-terminal-name]

[ (repeat | list) non-terminal-name+] %For repeat or list, a + can be
   added after the non-terminal name to indicate at least one.
```

opt The non-terminal optional. Short form: ?.

repeat Matches zero or more repetitions of the non-terminal. Short form: *. If one or more is required a + must be placed after the non-terminal name.

list Matches a possibly empty, comma-separated list of the non-terminal. Short form: ,. If one or more is required a + must be placed after the non-terminal name.

2.2 TXL Rule Language

Each TXL program must define a main rule or function which is applied to the entire input parse tree. User-defined rules and functions are then invoked from the main rule.

Variables

Variables are names that are bound to TXL trees whose type is explicitly given when the tree is introduced. There is a special anonymous variable which represents a nameless variable - this is just a placeholder and cannot be referenced.

[4]`http://www.txl.ca/nresources.html`

Listing 2.9: General form of TXL define statement

```
define name
        alternative1    %where alternatives are one or more non-terminals
           .
        | alternative2
        | alternative3
               .
               .
        | alternativeN
end define
```

Listing 2.10: General form of TXL redefine statement

```
redefine name
        ...                      % ... represents the original definition
        | alternative1
        | alternative2
        | alternative3
               .
               .
        | alternativeN
end redefine
```

Patterns

Patterns are made up of a sequence of non-terminals and variables which defines the type and shape of a particular tree which the rule is to match. In a rule the tree type of a pattern to match must be the same as the replacement tree type (Listing 2.12).

Built-in functions

There are many built-in functions in TXL - too many to list here - the important ones will appear in the implementation of the tasks. One important one is the extract function ˆ which is used to extract a pattern from one tree to create a collection of the matches.

Listing 2.11: General form of TXL function

```
function name
        replace [type]
                pattern
        by
                replacement
end function
```

Listing 2.12: TXL rule to resolve addition

```
rule resolveAddition
        replace [expression]
                A [number] + B [number]   % pattern matches a number
                    followed by an addition symbol
                                          % followed by another number.
                                          % A and B are variables bound to
                                              the numbers matched.
        by
                A [+ B]                           % use the built-in
                    function + to add the two numbers.
                                          % This creates a new tree of type
                                               number which
                                          % replaces the original.
end rule
```

3 Eclipse API

The Eclipse Platform [96], similar to Netbeans, is the basis for the Eclipse IDE. With refactoring being a requirement of the IDE, a way of transforming programs is needed. The package JDT [28], in Eclipse, is designed for the manipulation of Java programs. Though unlike Netbeans there isn't a package implementing a transformation language.

Programs for transformation and static analysis will be written in Java, unlike ANTLR and TXL where their own specialised language is used.

3.1 Parsing

The Eclise ASTParser is constructed with the Java Language Specification version as an argument - here we used JLS3 which is the latest specification. The parser accepts a char array of source code therefore a function to convert a file to a String is used in the example which is then converted to a char array (Listing 2.13). Finally a *CompilationUnit* object (the start symbol for a Java program) is obtained by calling the *createAST* method. The same few commands are used for all the Eclipse tasks.

CompilationUnit is a sub-class of *ASTNode* which contains a method *accept(Visitor v) : void.*

Listing 2.13: Java Eclipse - Parsing Java Source

```
char[] source = fileToString(new File(filename)).toCharArray();

ASTParser parser = ASTParser.newParser(AST.JLS3); //JLS3 - Java 1.5

parser.setSource(source);

CompilationUnit r = (CompilationUnit) parser.createAST(null /*
    IProgressMonitor */); // parse
```

Listing 2.14: Java Eclipse - Visitor class example

```
class MyVisitor extends ASTVisitor {
        public boolean visit(MethodDeclaration m) {
                System.out.println(m);
                return true;
        }
}
```

Listing 2.15: Java Eclipse - rewriting

```
ASTRewrite rewrite = ASTRewrite.create(r.getAST());
```

The *ASTVisitor* class can be extended then it's methods can be over-ridden to visit particular AST nodes (Listing 2.14). The *ASTVisitor* class provides a *visit(TYPE n) : boolean* and *endVisit(TYPE n) : void* for each type of node. If the former returns true all it's children will be visited after it has been visited. The *endVisit* methods visit nodes after the children have been visited. Two methods, *preVisit(ASTNode n) : void* and *postVisit(ASTNode n) : void*, can be used to visit all nodes regardless of type.

3.2 Re-writing

In order to re-write an AST in Eclipse the class *ASTRewrite* is used in conjunction with an AST visitor. An *ASTRewrite* object is created by calling the static method *create* with an ASTNode as a parameter.

4 JavaCC

Java Compiler Compiler (JavaCC)[5] is a parser generator for the Java language. It is similar to ANTLR [78] but can only output a parser in the Java language. A language grammar, in EBNF notation, is provided as input, and the JavaCC application outputs a parser, in the Java, for that language. The input language is not specifically Java, but the generated parser is Java. Building on the generated parser will allow static analysis and transformation operations to be performed on the specified language [75]. As we are concerned with program transformations we need to access an AST of the program to be transformed. JavaCC itself does not automate the building of AST's but a tool called JJTree[6] is used in-conjunction with JavaCC for this purpose. We will only consider the Java 1.5 language, for which grammars have been written and are freely available[7]. The available Java 1.5 grammar will allow the generation of a parser, which can be extended to create static analysis and transformation tools.

JavaCC's grammar language is similar to ANTLR and the Java grammar written by Julio Vilmar Gesser will be used. This already includes a complete AST definition unlike ANTLR where Java files for each node must be created. Therefore the focus here will be using the parser with an AST visitor to traverse the generated AST rather than adding actions to the grammar file (which can also be done in a similar way to ANTLR). Assuming this we will not discuss the grammar language in this document due to the availability of an existing complete grammar which generates a usable parser. Two lines of code (Listing 2.16, page 37) are needed to invoke the parser and start the tree traversal.

Listing 2.16: JavaCC - Parser invocation

```
CompilationUnit root = JavaParser.parse(System.in);

root.accept(VISITOR_OBJECT, null);
```

[5]https://javacc.dev.java.net/
[6]https://javacc.dev.java.net/doc/JJTree.html
[7]https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110

5 javac / Java API

Sun recently open-sourced it's Java Development Kit, which includes javac, the java compiler[8]. The Java Compiler is itself written in Java. Being a compiler for Java, javac includes a parser and tree generator for Java 1.5 and therefore no parser is needed to be generated using another tool. Java 1.6 also includes a Compiler API [93] though it's uses are limited to parsing a source file, viewing the AST (not modifiable), and compiling. To enable program transformations to be performed the internal javac's tree classes can used which provide access to AST fields. The use of internal classes could cause problems when javac is updated as internal classes are subject to change as they are not part of an official API. An advantage of using javac will be the accuracy of it's implementation - it's written by the same people that created Java.

[8]http://openjdk.java.net/groups/compiler/

6 Evaluation Tasks

The first half of this chapter described several tools for static analysis and program transformation tasks. In order to demonstrate their capabilities several tasks will be implemented with each tool. These tasks will be static analysis and/or program transformations and will allow each system to be evaluated and compared. The second half of this chapter describes these tasks and how they were implemented using each tool. Most tasks are fairly simple but will allow each tool to be tested and demonstrate it's abilities.

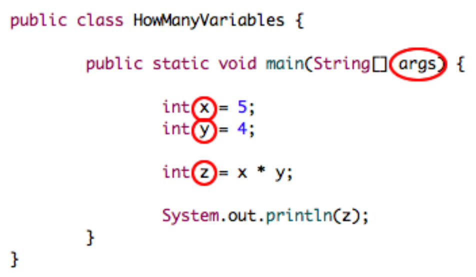
Tasks implementing graphs should output *DOT format* files, which can be used to define directed graphs. These files can be opened with graphviz [53] on Mac OS X to produce graphical representations of the graphs (example graph output Figure 1.8).

To evaluate the transformation and static analysis tools several tasks will be implemented using each tool. The tools will be compared in several categories: documentation, ease of use, efficiency, expressive power. The results may not be quantifiable due to the nature of the tools, but it should be clear which tools are a more appropriate for which tasks. It may be the case that some tasks cannot be implemented using all the tools due to tools being specialised in one area - for example the javac compiler is designed to parse and compile programs so it may not be possible to modify it in ways that some of the tasks require. Some tools may only be appropriate for one tasks, whereas others may be able to handle all the tasks.

Each tool should take a single Java source file and every tool should output the same result. There will be two types of evaluation task: transformation and static analysis. The static analysis tasks should provide some program metrics and information available by statically analysing the program. The transformation tasks should perform a rephrasing transformation; that is, they should take a Java program and output another Java program.

6.1 Count Program Variables

For a given Java source count all the variables declared in the program and display the count (Figure 2.1). The tool should take a Java source program as input and output a number.



```
public class HowManyVariables {

        public static void main(String[] args) {

                int x = 5;
                int y = 4;

                int z = x * y;

                System.out.println(z);
        }
}
```

Figure 2.1: Example program with it's four variables highlighted

ANTLR

A global variable, declared in the *members* section of the parser (Listing 2.17), is used to keep count of the number of variables in the program being parsed. Every time the variable declaration rule is matched the count is incremented (Listing 2.18). Finally, after the parsing is complete the *count* variable is output (Listing 2.19).

Eclipse API

A global variable is also declared (not shown in the code snippet) in the Eclipse API program in order to keep a count of the number of variables. The Eclipse API uses the Visitor Pattern to traverse the tree; the visitor interface provides a method *visit(VariableDeclarationFragment v) : boolean* which is visited whenever a variable is declared - it is here that the count variable is incremented (Listing 2.31).

JavaCC

The parser generated by JavaCC is very simliar to Eclipse API - it also uses the Visitor Pattern to traverse the AST. The signatures of the visiting interface methods are slightly different (Listing 2.21) and the AST is slightly different, otherwise the program is very similar (Listing 2.21, page 41). In the JavaCC parser the variable declaration node is called *VariableDeclarator* whereas in Eclipse it is called *VariableDeclarationFragment*.

javac / Java API

This is very similar to Eclipse and JavaCC - javac uses the Visitor Pattern to traverse the AST. In javac the node we are interested in is *VariableTree*. One difference with javac is that there are separate methods for each node type, rather than one overloaded method e.g. the method *visitVariable* is used to visit *VariableTree* nodes but in Eclipse the method *visit* is used to visit any node and uses polymorphism to decide which method to invoke.

TXL

The TXL program to count the number of declared variables is very different from the other tools. two built-in functions are used: the extract function and the length function. The extract function creates a collection of *variable_declaration* trees by extracting all the *variable_declaration* trees from the parsed program. The length, followed by print, functions are then applied to the newly created

Listing 2.17: ANTLR Member variable declaration

```
@parser::members {
        int count = 0;
}
```

Listing 2.18: ANTLR Variable Declarator Rule With Action

```
variableDeclarator
        : Identifier variableDeclaratorRest { count++; }
        ;
```

Listing 2.19: ANTLR CompilationUnit Rule with action Displaying the number of variables

```
compilationUnit
    @after {
        //The 'after' section executes after the complete rule is matched
           .
        //in this case after the complete program has been parsed.
        System.out.println(count);
     }
     :  annotations?
        packageDeclaration?
         importDeclaration*
        typeDeclaration*
     ;
```

Listing 2.20: Visitor method which prints out method invocations

```
public boolean visit(VariableDeclarationFragment v) {
        count++;
        return false;
}
```

Listing 2.21: Java Program to Count Variables with JavaCC Parser

```
public void visit(VariableDeclarator n, Object arg) {
        count++;
        super.visit(n, arg);
}
```

Listing 2.22: Javac / Java API visitor method to count variables

```
public Void visitVariable(VariableTree arg0, Void arg1) {
       count++;
       return super.visitVariable(arg0, arg1);
}
```

Listing 2.23: TXL Program to count the number of variables declared in a Java program

```
function main
        replace [program]
                P [program]
        construct AllVars [repeat variable_declaration]
                _ [^ P]
        construct N [number]
                _ [length AllVars] [print]
        by
                %nothing
end function
```

list variable. This results in the number of variables being displayed. The program is then replaced by nothing (as the output is created via the *print* function) (Listing 2.23).

Conclusion

Eclipse API, JavaCC and javac are all very similar as they all use the Visitor Pattern to traverse the AST. Therefore these need only to increment a variable whenever a variable declaration node is visited. The only differences are the structure of the ASTs and different Visitor interfaces.

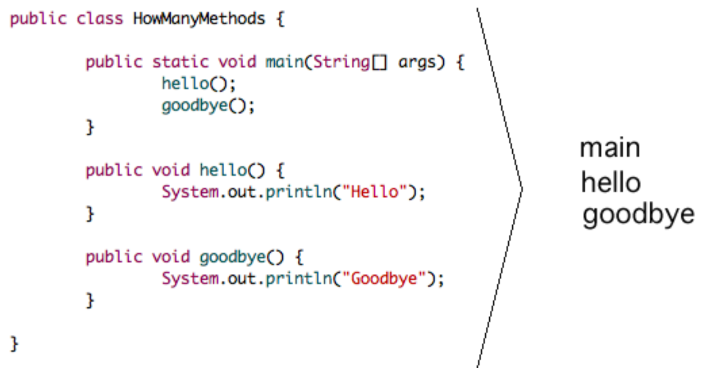
ANTLR requires some additions to the Java grammar in the form of actions - the action when a variable is to increment a global variable, the action after parsing is to print out the variable value.

TXL takes a very different approach as it doesn't work in the same way as the others. The built-in *extract* function is used to create a collection of all the *variable_declaration* trees found in the program. The *length* function is then applied followed by the *print* function to display the length of the collection.

The simplest programs to implement included Eclipse, JavaCC and javac with the use of their visitors making it much simpler than ANTLR or TXL.

6.2 List Program Methods

For a given Java source display a list of all the methods declared in the source file. The tool should take a Java source program as input and output each method name on a new line (Figure 2.2).



```
public class HowManyMethods {

        public static void main(String[] args) {
                hello();
                goodbye();
        }

        public void hello() {
                System.out.println("Hello");
        }

        public void goodbye() {
                System.out.println("Goodbye");
        }

}
```

```
main
hello
goodbye
```

Figure 2.2: Example program with it's methods listed

ANTLR

Listing program methods consists of parsing the source code and outputting the name of a method whenever one is declared. At each rule defining a method an action is added which prints the method name (Listing 2.24). ANTLR allows rules to be given labels which can be referred to, in the target language, in the rule action. In the rule *methodDeclaration* the Identifier is labelled *m* and the text of the Identifier is referred to in the Java code as *$m.text*. In the Java grammar there are two rules which deal with method declarations: *memberDecl* and *methodDeclaration*.

Eclipse API

The Eclipse AST has a node called *MethodDeclaration* - therefore this visitor method is over-ridden to print out the name of the method declaration (Listing 2.25).

JavaCC

The JavaCC generated parser is very similar to Eclipse - the name of the AST node is also the same - the only difference being in the visitor interface and node attributes (Listing 2.26).

javac / Java API

The javac visitor method (Listing 2.27, page 44) is similar to Eclipse and JavaCC. The method *get-MethodSelect* is used here to retrieve the name of the method.

TXL

The list method program transforms a Java program into a list of methods defined in that program (Listing 2.28). The start symbol is redefined as either a Java program or zero or more methodname non-terminals.

The methodname non-terminal is also redefined to include a newline symbol in the output, so that each method name is on a newline.

The main function matches a program and uses the extract function (^) to extract all methodname non-terminals.

Conclusion

Eclipse API, javac, JavaCC were again similar in this task with the only differences being in their ASTs and visitor interfaces. Implementing the task in ANTLR was slightly simpler than the previous task as we did not need to store anything - we just output the name of a method when found. This involved adding an action to put out the method name in the two appropriate rules: *memberDecl* and *methodDeclaration*.

Listing 2.24: ANTLR Rule - memberDecl

```
memberDecl
        :       genericMethodOrConstructorDecl
        |       methodDeclaration
        |       fieldDeclaration
        |       'void' m = Identifier voidMethodDeclaratorRest
                        { System.out.println($m.text); } //Action: Print
                            Identifier Name
        |       Identifier constructorDeclaratorRest
        |       interfaceDeclaration
        |       classDeclaration
        ;
methodDeclaration :
                type m = Identifier methodDeclaratorRest
                        { System.out.println($m.text); } //Action: Print
                            Identifier Name
        ;
```

Listing 2.25: Java program to list a Java programs methods

```
public boolean visit(MethodDeclaration v) {
        System.out.println(v.getName());
        return false;
}
```

Listing 2.26: Java Program to list the methods declared in a Java program using JavaCC Parser

```
public void visit(MethodDeclaration n, Object arg) {
        System.out.println(n.name);
        super.visit(n, arg);
}
```

Listing 2.27: Javac / Java API visitor method to output method names

```
public Void visitMethodInvocation(MethodInvocationTree arg0, Void arg1) {
        System.out.println(arg0.getMethodSelect());
        return super.visitMethodInvocation(arg0, arg1);
}
```

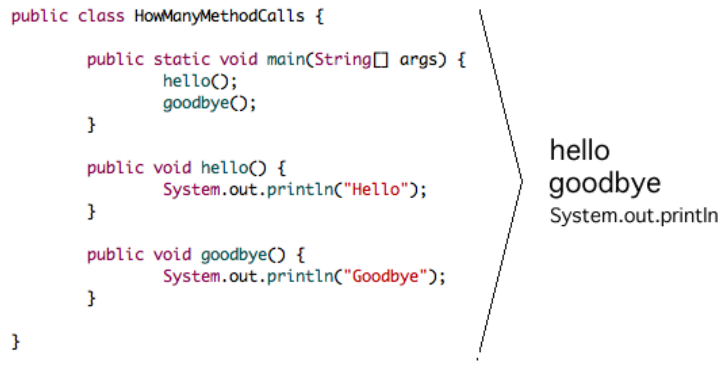
Listing 2.28: TXL program to display a Java programs methods

```
redefine program                          % redefine program to be either a
    program
        ... | [repeat method_name]% or a collection of method_name trees
end redefine

redefine method_name
        ... [NL]                          % redefine method_name to include
            a newline after it in the output.
end redefine

function main
        % match a program
        replace [program]
                P [program]
        % construct a new variable with the type [repeat method_name]
        construct AllMethods [repeat method_name]
        % constructs the new from by using the extract function
                _ [^ P]
        by
        % replaces the match with the newly constructed variable
                AllMethods
end function
```

The TXL program is similar to the previous task but required the redefinition of two non-terminals something that is very different from the other tools.

6.3 List Program Method Calls

For a given Java source display a list of all the method calls in the source file. The tool should take a Java source program as input and output each method call on a new line (Figure 2.3). Not necessarily all the methods listed will actually be called when the program is executed because during static analysis the program is not executed.



```
public class HowManyMethodCalls {

        public static void main(String[] args) {
                hello();
                goodbye();
        }

        public void hello() {
                System.out.println("Hello");
        }

        public void goodbye() {
                System.out.println("Goodbye");
        }

}
```

```
hello
goodbye
System.out.println
```

Figure 2.3: Example program with it's method calls listed

ANTLR

Again the ANTLR grammar has some actions added to it which output the name of the methods. Actions need to be added in two places - one for the actual method name (Listing 2.29, page 47) and one for the possible suffix (Listing 2.30, page 47). Both the actions in the grammar simply print out the identifier name.

Eclipse API

The Eclipse visitor simply visits a *MethodInvocation* node and prints out, if it isn't null, the method's expression (known in ANTLR as the suffix) and the method name.

JavaCC

JavaCC is similar to Eclipse but the method's expression is known as *scope*. It also simply prints out the method scope and method name in the visitor method (Listing 2.33, page 47).

javac / Java API

The visitor for javac uses the *visitMethod* method and prints out the method's name which unlike ANTLR, Eclipse and JavaCC already includes the suffix/expression/scope part (Listing 2.33, page 47).

TXL

In the TXL program to list method calls a new non-terminal, *method_invocation*, is defined which matches the pattern of a method call. The program start-symbol is redefined to be either a program or a collection of *method_invocation* trees allowing the output of the TXL program to be the list of method calls.

Conclusion

Listing program method calls is very similar to listing methods the only difference being the node visited in the AST, or rule matched (in the case of ANTLR and TXL). In TXL we needed to define a new non-terminal *method_invocation* which matches a method call and redefine *reference* non-terminal to match a *method_invocation* - this made the task slightly more complicated than the previous task.

Listing 2.29: ANTLR Rule - primary - to print method call

```
primary
    :   parExpression
    |   nonWildcardTypeArguments
        (explicitGenericInvocationSuffix | 'this' arguments)
    |   'this' ('.' Identifier)* (identifierSuffix)?
    |   'super' superSuffix
    |   literal
    |   'new' creator
     |   a = Identifier {System.out.print($a.text);} ('.' b = Identifier
        {System.out.print("." + $b.text);} )* arguments
                {System.out.println();}
    |   Identifier ('.' Identifier)* (identifierSuffix)?
    |   primitiveType ('[' ']')* '.' 'class'
    |   'void' '.' 'class'
        ;
```

Listing 2.30: ANTLR Rule - explicitGenericInvocationSuffix - to print method call

```
explicitGenericInvocationSuffix

        :       'super' superSuffix
        |   i = Identifier arguments { System.out.println($i.text); }
        ;
```

Listing 2.31: Visitor method which prints out method invocations

```
public boolean visit(MethodInvocation v) {
        if(v.getExpression() != null)
                System.out.print(v.getExpression() + ".");

        System.out.println(v.getName());
        return false;
}
```

Listing 2.32: Java program to list method calls using the JavaCC parser

```
public void visit(MethodCallExpr n, Object arg) {
        System.out.println(n.scope + "." + n.name);
        super.visit(n, arg);
}
```

Listing 2.33: Javac / API visitor method to output method names

```
public Void visitMethod(MethodTree arg0, Void arg1) {
            System.out.println(arg0.getName());
            return super.visitMethod(arg0, arg1);
}
```

Listing 2.34: TXL Program to list the methods calls in a Java program

```
redefine program
        ...                                               % redefine a
           program to be a program
        | [repeat method_invocation]    % or a collection of
           method_invocation trees
end redefine

define method_invocation        % define a method_invocation non-terminal
                                               % include output
                                                  terminals (SPOFF and
                                                  NL) to affect the
                                                  formatting
        [qualified_name] [SPOFF] [method_argument] [NL]
end define

redefine reference              % redefine a reference to be
        [method_invocation]     % either a method_invocation
        | ...                           % or a reference.
end redefine

function main
        replace [program]
                P [program]
        construct  AllMethodCalls [repeat method_invocation]
                _ [^ P] % use the extract function to construct a
                    collection of all method_invocation trees
        by
                AllMethodCalls
end function
```

Listing 2.35: Example program A (compare to 2.36)

```
public class Foo {
        public static void main(String[] args) {
                if(false) {
                        System.out.println("Hello");
                }else{
                        System.out.println("Goodbye");
                }
        }
}
```

Listing 2.36: Example program B (compare to 2.35)

```
public class Foo {
        public static void main(String[] args) {
                if(true) {
                        System.out.println("Hello");
                }else{
                        System.out.println("Goodbye");
                }
        }
}
```

6.4 Call Hierarchy Graph

A call hierarchy graph is a directed graph representing the method calls in a program from the methods that called them. The nodes of the graph represent methods in the program, and the edges represent method calls. For example the call graph for the simple test program (Listing 1) is shown in figure 3.7. The output is a graph off all possible method calls, but not necessarily all actual method calls due to this being a static analysis operation which will not execute the code. Listings 1 and 2 are semantically different but will produce the same call graph when analysed statically.

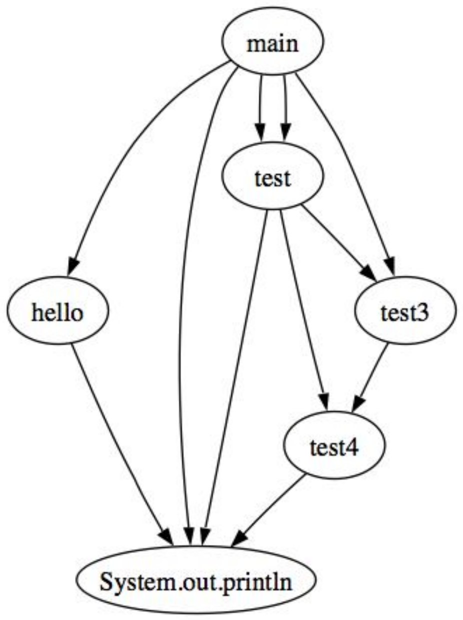


Figure 2.4: Call Graph of Listing 1

Listing 2.37: ANTLR Rule - methodInvocation

```
methodInvocation
@init{
        //the 'init' section is executed before the rule is matched
        //useful for initialising variables
        String identifier = "";
}
@after{
        currentNodes.add(new Node(identifier));
}
:
        i = Identifier { identifier += $i.text; }
                ('.' i = Identifier { identifier += "." + $i.text; } )*
                        arguments
;
```

Listing 2.38: ANTLR Java argument list rule

```
arguments : '(' expressionList? ')' ;
```

ANTLR

In order to produce a graph all method calls need to be linked to the methods that they were called from. Therefore there are two places were actions need to be added to the grammar - method declaration (same as for method listing) and method invocation rules. Graphs will be output in the *dot format.*

A method call is defined as 'an Identifier followed by zero or more fullstops and Identifiers followed by arguments' where arguments are defined as left parenthesis followed by a possible list of expressions followed by right parenthesis' (Listing 2.38). Everytime a method identifier is matched the fully qualified name of that identifier is added to a set of current nodes. The rule builds up the fully qualified identifier name, and after the complete rule is matched the identifier is added to set of current nodes (Listing 2.37).

In each method declaration rule an edge is created between that method and each of the current nodes; the current nodes set is then emptied (Listing 2.39, 2.40).

After the complete source is parsed the dot format graph data is output, using an 'after' section of the *compilationUnit* rule.

Eclipse API

The Eclipse visitor for producing a call graph implements the two methods for visiting *MethodDeclaration* and *MethodInvocation* nodes (Listing 2.41, page 51). In the *MethodDelcaration* method a global variable containing the name of the current node is updated. In the *MethodInvocation* method an edge is printed out consisting of the current node and the name of the method being invoked.

JavaCC

The JavaCC visitor works in much the same way as the Eclipse one by implementing methods which are visited during method invocation and declaration (Listing 2.42, page 52).

javac / Java API

The javac visitor works in exactly the same way as Eclipse and JavaCC (Listing 2.43, page 52).

TXL

The TXL program to produce a call graph is very different from the other tools (Listings 2.44, page 53 and 2.45, page **??**). It is longer and more complicated. Like the program to list method calls (Listing 2.34, page 48) the new non-terminal *method_invocation* is defined and *reference* is redefined.

Listing 2.39: ANTLR Creating edges between methods

```
memberDecl
        :       genericMethodOrConstructorDecl
        |       methodDeclaration
        |       fieldDeclaration
        |       'void' m= Identifier voidMethodDeclaratorRest
                        {
                                for(Node n : currentNodes) {
                                        edges.add(new Edge(n.getName(),
                                           $m.text));
                                }

                                currentNodes.clear();
                        }
        |       Identifier constructorDeclaratorRest
        |       interfaceDeclaration
        |       classDeclaration
;
```

Listing 2.40: ANTLR Creating edges between methods

```
methodDeclaration
        :       type m= Identifier methodDeclaratorRest
                {

                        for(Node n : currentNodes) {
                                edges.add(new Edge(n.getName(),$m.text));
                        }

                        currentNodes.clear();
                }

;
```

Listing 2.41: Two visitor methods which produce a call graph

```
public boolean visit(MethodDeclaration v) {
        currentNode = v.getName().toString();
        return true;
}

public boolean visit(MethodInvocation v) {

        String n = "";
        if (v.getExpression() != null)
        n = v.getExpression().toString() + ".";

        n += v.getName().toString();

        System.out.println("\t\"" + currentNode + "\" -> \"" + n + "\";")
            ;

        return true;
}
```

Listing 2.42: Java program to create a call hierarchy graph using the JavaCC parser

```
public void visit(MethodCallExpr n, Object arg) {
        String name = n.name;
        if(n.scope != null) name = n.scope + "." + name;

        System.out.println("\t\"" + currentMethod + "\" -> \"" + name + "
            \"");
        super.visit(n, arg);
}

public void visit(ObjectCreationExpr n, Object arg) {
        String name = n.type.name;
        if(n.scope != null) name = n.scope + "." + name;

        System.out.println("\t\"" + currentMethod + "\" -> \"" + name + "
            \"");
        super.visit(n, arg);
}

public void visit(MethodDeclaration decl, Object arg) {
        currentMethod = decl.name;
        super.visit(decl, arg);
}

public void visit(ConstructorDeclaration decl, Object arg) {
        currentMethod = decl.name;
        super.visit(decl, arg);
}
```

Listing 2.43: Java API Visitor to produce a call graph

```
        public Void visitMethod(MethodTree arg0, Void arg1) {
            currentMethod = arg0.getName().toString();
            return super.visitMethod(arg0, arg1);
        }

        public Void visitMethodInvocation(MethodInvocationTree arg0, Void
             arg1) {

           System.out.println("\t\"" + currentMethod + "\" -> \"" + arg0.
               getMethodSelect().toString() + "\";\n");

            return super.visitMethodInvocation(arg0, arg1);
        }
```

Listing 2.44: TXL grammar definitions to produce a call graph in dot format for a Java program

```
%define graph

define method_invocation
        [qualified_name] [SPOFF] [method_argument] [NL]
end define

redefine reference
        [method_invocation]
        | ...
end redefine

define graph
        'digraph [id] '{ [NL]
        [IN] [repeat edge]
        [EX] '}
end define

define edge
        [stringlit] [SPOFF] '-> [SPON] [stringlit] ; [NL]
end define
```

Additionally *graph* and *edge* non-terminals are defined - these together provide the ability in the rules section of the program to create a graph.

In the rules section of the TXL program two rules are defined: *find_methods* and *find_calls*. The rule *find_methods* finds all the *method_declaration* trees in the program, deconstructs them and invokes the *find_calls* rule, on the method's body, which then deconstructs the *method_invocation* trees, inside the body, to create edges between the *method_declaration* and the *method_invocation*. The edges are stored in a global lookup table which is declared in the *main* function, using the built-in *export* and *import* functions.

Conclusion

The call hierarchy graph task is more complicated than the first tasks especially with TXL. Using all the tools, except TXL, involves storing the current method definition being visited and then creating an edge between that and the invocations visited after the method definition. ANTLR is only slightly different, in that it doesn't use visitors, but actions are added to the grammar to do the same thing. Using TXL we must first define a graph and the implement rules to match method definitions and the invocations with them which turned out to be a much more difficult task than using the other tools.

Listing 2.45: TXL transformation rules to produce a call graph in dot format for a Java program

```
%being transformation rules

function main
        export Edges [repeat edge] _
        replace [program]
                P [program]
        construct NewP [program]
                P [find_methods]
        construct Graph [graph]
                'digraph G '{ Edges '}
        construct Output [graph]
                Graph [print]
        by
                %nothing
end function

rule  find_methods

        replace $ [method_declaration]
                M [method_declaration]
        deconstruct M
                Modifier [repeat modifier] Type [type_specifier]
                   Declarator [method_declarator] Throws [opt throws]
                   Body [method_body]
        deconstruct Declarator
                Name [method_name] '( Params [list formal_parameter] ')
                   Dim [repeat dimension]
        construct _ [method_body]
                Body [find_calls Name]
        by
                M
end rule

rule find_calls Name [method_name]
        import Edges [repeat edge]

        replace $ [method_invocation]
                Invocation [method_invocation]
        deconstruct Invocation
                B [qualified_name] _ [method_argument]
        construct S1 [stringlit]
                _ [quote Name]
        construct S2 [stringlit]
                _ [quote B]
        construct Edge [edge]
                S1 '-> S2 ';
        export Edges
                Edges [. Edge]
        by
                Invocation
end rule
```

Listing 2.46: HelloWorld

```
public class HelloWorld {

        public static void main(String[] args) {
                System.out.println("Hello World");
        }

}
```

6.5 AST Graph

A finite, labelled, directed graph representing a program where each node represents a language construct and children are parameters of those constructs. Tools may represent the AST of a program differently depending on how the implement their AST, but each program should return a similar AST. Figure 2.5 represent an AST for the program 2.46. The root of all Java AST's is the *compilation unit* - the start symbol.

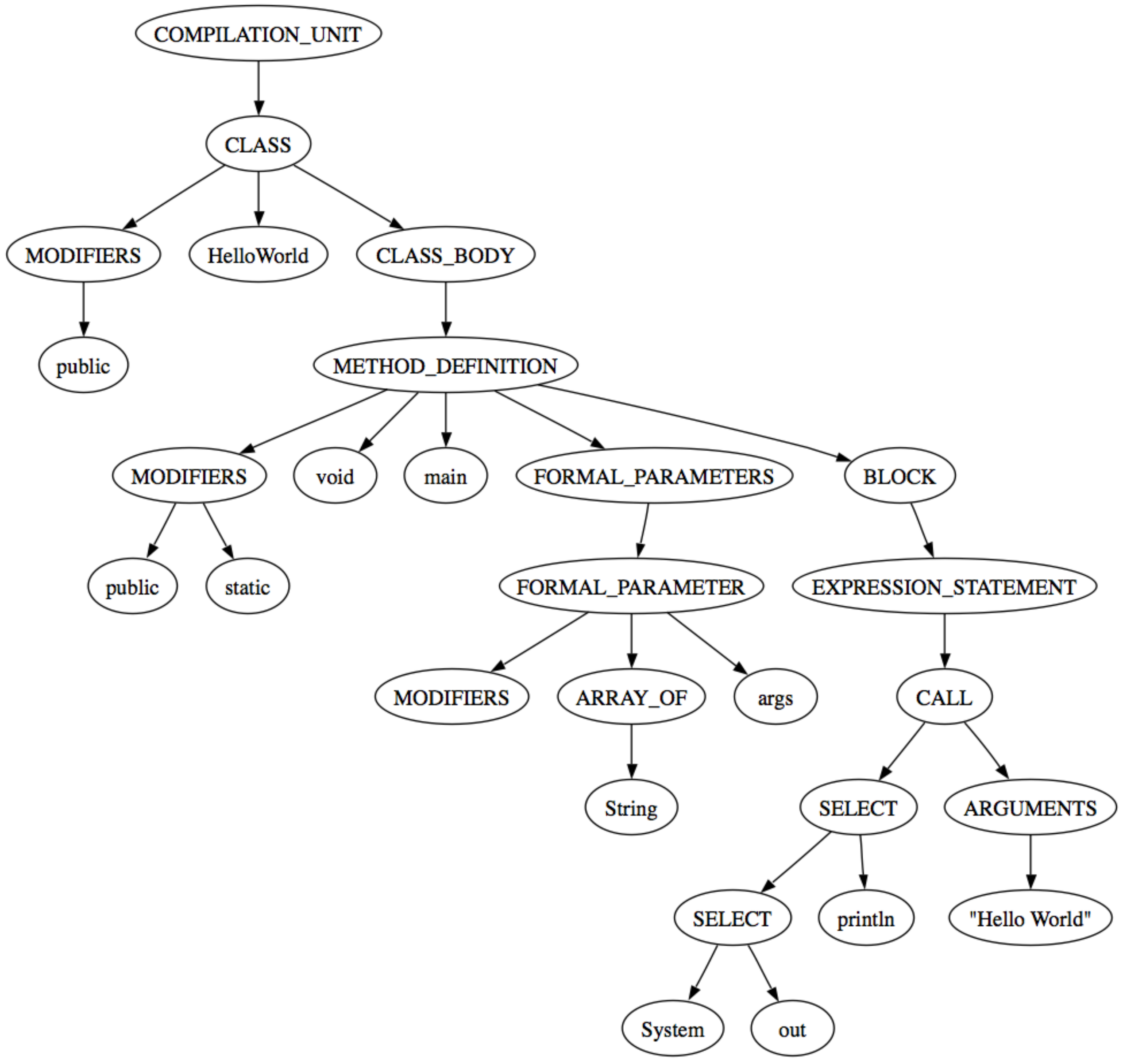


Figure 2.5: Hello World (Listing 2.46) AST

Listing 2.47: Java Program to invoke ANTLR lexer and parser and output a DOT format graph of the AST

```
/* StringTemplate - modified to effect output graph style */
StringTemplate _treeST =
        new StringTemplate(
          "digraph {\n" +
          "  $nodes$\n" +
          "  $edges$\n" +
        "}\n");

StringTemplate _edgeST =
          new StringTemplate("$parent$ -> $child$;\n");

/* JavaASTParser.compilationUnit_return is the return type of the
    compilationUnit rule */
JavaASTParser.compilationUnit_return result = parser.compilationUnit();

Tree t = (Tree)result.getTree(); //get the tree

DOTTreeGenerator dot = new DOTTreeGenerator(); //create a DOT format
    generator

System.out.println(dot.toDOT(t, new CommonTreeAdaptor(), _treeST, _edgeST
    )); //print
```

ANTLR

In order to create an AST for a given program tree commands need to be added to the grammar[9]. The option to output an AST after parsing is then available. The output AST can be converted into a dot file via the use of an ANTLR class *DOTTreeGenerator* which takes an AST and returns a dot format string.

When the output option is set to AST the parser methods for each rule return an object which contains an AST. Some changes are made to the invocation program (Listing 2.7) which get the tree from a compilation unit, create a DOTTreeGenerator and output the dot format tree (Listing 2.47).

Eclipse API

Eclipse is very different from ANTLR for this task. In ANTLR tree building commands are written in the grammar. Eclipse already creates an AST which is traversed by a visiting as in the the previous task.

The generic *visitNode(ASTNode v)* method is implemented which is visited for every node on the tree, no matter what type. Two helper methods have been implemented: *printNodeName* and *printEdge*. The method *printNodeName* simply prints either the name of the node (i.e. method name or variable name etc) or the type of the node. The method *printEdge* prints out an edge consisting of two nodes hash codes which uniquely identify them (Listing 2.48, page 57).

javac / Java API

javac is based around the same visitor concept as Eclipse but wasmore complicated for this task as AST nodes do not have a reference to their parent. Therefore, with javac, we had to over-ride all the visitor methods and not just a generic visit all method. This allowed us to create an edge between every node and its children.

[9]Java 1.5 ANTLRv3 Grammar with Tree Rules by John Ridgway

Listing 2.48: Methods called by each node visitor to build an AST graph

```
private void printNodeName(ASTNode v) {

        if(v != null) {
                String name = "";
                if(v instanceof SimpleName
                        || v instanceof StringLiteral
                        || v instanceof NumberLiteral
                        || v instanceof BooleanLiteral
                        || v instanceof Modifier
                        || v instanceof PrimitiveType) {

                         UUID uuid = java.util.UUID.randomUUID();

                         name =  ASTNode.nodeClassForType(v.getNodeType()
                             ).getSimpleName();

                        System.out.println("\t" + v.hashCode() + " [label
                            =\"" + name + "\"];");

                        name = v.toString();

                        System.out.println("\t" + uuid.hashCode() + " [
                            label=\"" + name + "\"];");

                        System.out.println("\t" + v.hashCode() + " -> " +
                             uuid.hashCode() + ";");


                }else{

                        name =  ASTNode.nodeClassForType(v.getNodeType())
                            .getSimpleName();

                        System.out.println("\t" + v.hashCode() + " [label
                            =\"" + name + "\"];");
                }


        }
}

private void printEdge(ASTNode a, ASTNode b) {
        if(a != null && b != null)
                System.out.println("\t" + a.hashCode() + " -> " + b.
                    hashCode() + ";");
}

public boolean visitNode(ASTNode v) {

        printNodeName(v.getParent());
        printNodeName(v);
        printEdge(v.getParent(), v);

        return true;
}
```

Listing 2.49: javac visitor methods for creating an AST graph

```
private void makeNode(int a, String label) {
        graph.append("\t\"" + a + "\" [label=\"" + label + "\"];\n");
}

private void makeNode(Tree t) {
        makeNode(t, t.getKind().toString());
}

private void makeNode(Object o, String s) {
        makeNode(o.hashCode(), s);
}

private void makeEdge(int a, int b) {
graph.append("\t\"" + a + "\" -> \"" + b + "\";\n");
}

private void makeEdge(Tree a, Tree b) {
        if(a != null && b != null) {
                makeNode(a);
                makeEdge(a.hashCode(), b.hashCode());
                makeNode(b);
        }
}

private void makeEdge(Tree a, String b) {
        if(a != null) {
                makeNode(a);
                graph.append("\t\"" + a.hashCode() + "\" -> \"" + b + "
                    \";\n");
        }
}

private void makeEdge(String a, String b) {
        graph.append("\t\"" + a + "\" -> \"" + b + "\";\n");
}

private void makeEdge(Tree a, List<? extends Tree> list) {
        if(list.size() > 0)
                for(Tree t : list)
                        makeEdge(a, t);
}

private void makeEdge(Tree a, Set<? extends Modifier> list) {
        if(list.size() > 0)
                for(Modifier t : list)
                        makeStringEdge(a, t.toString());
}

private void makeStringEdge(Tree a, String s) {
        UUID uuid = java.util.UUID.randomUUID();
        makeNode(uuid.hashCode(), s);
        makeEdge(a.hashCode(), uuid.hashCode());
}
```

Listing 2.50: TXL expression evaluator grammar AST modifications

```
redefine program
        ... |
        ( EXP [expression] )
end redefine

redefine expression
        ... |
        ( ADD [expression] [term] ) |
        ( SUB [expression] [term] )
end redefine

redefine term
        ... |
        ( MUL [term] [primary] ) |
        ( DIV [term] [primary] )
end redefine

redefine primary
        ... |
        ( NUM [number] ) |
        [expression]
end redefine
```

TXL

TXL does not create an AST of the program transformed. As it is not designed to do this a TXL program must be written to do this manually. The technique is demonstrated here using a TXL program to evaluate expressions[10].

The original grammar for the expression evaluator is redefined to include new versions of each non-terminal. Each non-terminal can be either the original mathematical expression or an AST version of that node (Listing 2.50). Notice that in *expression* and *term* there are two possible trees are listed for each. The non-terminal *primary* can also be an *expression* without parenthesis in the AST version of the program.

The format of an AST node is *(TYPE child1 [child2])* (type *NUM* only has one child).

This program works in the same way as the original expression evaluator (Listing B.1), but the rules are modified to replace the original mathematical expression with its AST equivalent, instead of resolving the expression (Listing 2.51).

The same technique can be applied to Java grammar and rules. Listing 2.52 shows an example of defining and creating a *COMPILATION_UNIT* node. The output of such a program would be of the form *(TYPE child1 child2 ... childN).*

Conclusion

The AST graph turned out to be the most completed task for some of the tools. ANTLR became the easiest tool to use for this due to John Ridgway's Java grammar with tree building rules and ANTLR's built in AST to *DOT* format convertor.

The implementation in Eclipse was fairly simple using a visitor to visit every node and create a graph edge between it and its parent. javac, although based around the same visitor concept as Eclipse, was more complicated due to AST nodes not have a reference to their parent. Therefore, with javac, we had to over-ride all the visitor methods and not just a generic 'visit all' method. This allowed us to create an edge between every node and its children.

We found that TXL is not the tool to use for this task as it does not produce an AST. It is possible though to manually build an AST by applying transformation to a Java program but it is non-triivial and we have only given a sample of how we believe it could be implemented.

[10]Expression Evaluator from TXL Examples package (Listing B.1)

Listing 2.51: TXL expression evaluator rules AST

```
rule main
    replace [program]
        E [expression]
    construct  NewE [expression]
        E [resolveAddition] [resolveSubtraction] [resolveMultiplication]
          [resolveDivision] [resolveParentheses] [resolvePrimary]
    where not
        NewE [= E]
    by
        ( EXP NewE )
end rule

rule resolveAddition
    replace [expression]
        N1 [expression] + N2 [term]
    by
        ( ADD N1 N2 )
end rule

rule resolveSubtraction
    replace [expression]
        N1 [expression] - N2 [term]
    by
        ( SUB N1 N2 )
end rule

rule resolveMultiplication
    replace [term]
        N1 [term] * N2 [primary]
    by
        ( MUL N1 N2 )
end rule

rule resolveDivision
    replace [term]
        N1 [term] / N2 [primary]
   by
        ( DIV N1 N2 )
end rule

rule resolveParentheses
    replace [primary]
        ( E [expression] )
    by
         E
end rule

rule resolvePrimary
        replace [primary]
                N [number]
        by
                ( NUM N )
end rule
```

7 Conclusion

After undertaking several tasks using each tool we found that some tools are easier to use than others and there are tools which we would rather not use for our purpose of developing a plagiarism detection system.

One advantage that Sun's Java Compiler has over the other tools is that it should always be an accurate implementation of Sun's Java Specification. Though this would seem to be correct it is not necessarily the case as pointed out by Bruce Eckel [33], in 2006, who found a that the Eclipse JDT compiler handle a certain case using generics more accurately than javac. Even so we believe javac can be relied on for most cases more than some of the other tools discussed.

Tools that rely on grammars written by people not involved with Sun on the Java language can contain errors. One such error was found in the Java grammar for TXL during the evaluation of the tool (Figure D.1, page 121). These can cause problems when wanting a tool to perform static analysis or program transformation tasks because the parser needs to be correct.

The disadvantage of using Sun's Java Compiler for static analysis or program transformation is that it was not designed for the job and must be co-erced into doing so, for example in order to make changes to the AST produced by javac we must using javac's internal classes rather than the public classes that are part of the Java 1.6 Compiler API which provides a read-only tree. We hope this may change in future versions of the Compiler API.

Another problem with systems which rely on user written grammars is that more than one person can interpret the specification differently and implement different grammars. There are two Java grammars for ANTLR which both produce different ASTs which means that some people may use one while others use the other, though neither may be better than the other [91]. Different systems also do not produce the same AST - there is no 'standard' Java AST - so for example Eclipse uses a different AST than javac. Though after deciding on a tool to use this doesn't present a problem unless you have to change to another system were the AST is different.

If we were interested in analysing or transforming languages other than Java we would suggest using ANTLR as it can work with any language for which a grammar is defined. We did not like the mixture of grammar rules and programming logic (in the form of actions) inside the ANTLR grammar but if a parser is built with an AST it would allow a separation of these two. We did not have time to write a grammar to produce a full AST and a pretty-printer to output the code again so we were just concerned with inserting actions in to the grammar. We liked JavaCC more than ANTLR for the the simple reason that we downloaded JavaCC along with a pre-written grammar and Java based AST visitor implementation. Most likely if the same was available for ANTLR we would have prefered ANTLR as it can support more languages and it's grammar seems tidier. The JavaCC grammar is very difficult to understand and seems very untidy - the embedded Java actions (used to create an AST) can confuse and it is difficult to understand.

One feature we liked in ANTLR was the built-in class to produce *DOT* tree files which made producing the AST graph simpler using ANTLR than any of the other tools[11].

The visitor pattern is an easy way to traverse ASTs in Eclipse, javac and JavaCC and this made them very similar. But we exclude javac as a pratical tool for static analysis and program transformation as it was not designed for this task. Adapting javac was not as easy as we had envisaged though we did learn some useful things about the java compiler.

We liked JavaCC but would not want to use it to develop parsers with it's grammar language which must be ladened with actions to build an AST. There is also a lack of JavaCC documentation and it's website is very sparse - this made it difficult to understand JavaCC's grammar language.

The Eclipse JDT is an accurate implementation of the Java specification and Eclipse is a big project supported by IBM. We liked the ease of implementing visitor methods to traverse trees and the rewrite system works well. The Eclipse JDT is part of the excellent Eclipse IDE. With Eclipse there was no playing around with grammars, which may or may not be correct. There is not much documentation on Eclipse but we found it easy enough to decipher the use of visitors to traverse ASTs. One area were there could be more documentation is program transformation via the re-writing classes.

We have decided that TXL is 'not our cup of tea' due to being a functional programming language for which we are not used to. TXL is interesting though and we believe it is very powerful for certain tasks but we do not have enough experience to be able to make as much use of the system at this stage. There is a good amount of documentation available via TXL's website which was very useful in helping us understand how to use the it.

[11]thanks to a grammar with tree building functions by John Ridgway

It is not an easy task to say which is the best tool to use as everyone has different preferences. We are more familiar with Java programming and lean towards using Java based system to transform Java programs. We liked Eclipse JDT and will now try using it for a bigger task - a plagiarism detection system - which will allow us to see if we were correct in choosing this as our favourite system.

Listing 2.52: TXL expression evaluator rules AST

```
redefine program
        ... | ( COMPILATION_UNIT [package_declaration] )
end redefine

rule translateProgram
        replace [program]
                P [package_declaration]
        construct NewProgram [program]
                ( COMPILATION_UNIT P )
        by
                NewProgram
end rule
```

Chapter 3

Survey of Plagiarism Detection Techniques

There is much research in the area of plagiarism detection for both natural languages and programming languages. We are only interested in the detection of plagiarism in sets of program source code, specifically Java, though some techniques appropriate for natural language could be applied to programming languages. In 2000, Clough [20] surveyed a variety of plagiarism detection systems and techniques for natural and programming languages. We discuss some of the current techniques in this chapter.

1 String pattern matching

One of the simplest method employed to detect plagiarism in both natural language and programming languages is string pattern matching. This technique takes into account only the actual words or letters used in a document and not semantics. A program such as the UNIX *diff* utility[1] can compare two source texts and give a percentage similarity by solving the longest common subsequence problem [47]. Figure 3.1 (page 64) shows an example output from *diff* for programs 1.1 (page 10) and 1.2 (page 10). The output includes lines that are *different* between the two files.

```
1c1
< public class VarChangeA {
---
> public class VarChangeB {
3,5c3,5
<  public static void main(String[] args) {
<  for(int i = 1; i <= 10; i++)
<             System.out.println(i + ": " + factorial(i));
---
>  public static void main(String[] n) {
>  for(int b = 1; b <= 10; b++)
>             System.out.println(b + ": " + f(b));
8,9c8,9
<  public static int factorial(int n) {
<  if(n <= 1)
---
>  public static int f(int a) {
>  if(a <= 1)
12c12
<  return n * factorial(n - 1);
---
>  return a * f(a - 1);
```

Figure 3.1: Output of *diff* for programs 1.1 (page 10) and 1.2 (page 10)

[1]see the man page at `http://unixhelp.ed.ac.uk/CGI/man-cgi?diff` for more details

String pattern matching techniques does not require knowledge of the language being analysed and do not take into account structural changes. An advantage of this being that such a system could be used to detect similarity between an type of text document (as *diff* does). But it may not be as accurate as a system that takes into account structural changes which are an important plagiarism technique for programming languages. For example if a comparison technique was to compare the frequency of characters in two files simple techniques such as changing variable names would easily thwart the system.

There exists many string comparaison algorithms [21] and many are useful for finding similarities in text though not all will be suitable for detecting plagiarism in program source code. Two such algorithms which have been applied to source code similarity problems are Levenshtein distance [60] and Smith-Waterman algorithm [89, 48].

1.1 Levenshtein Distance

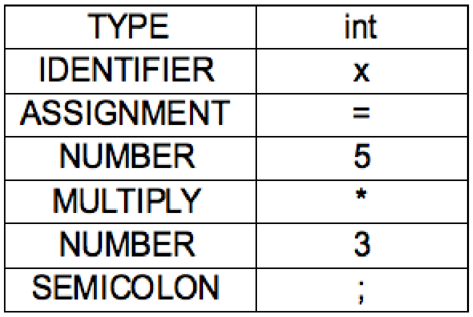
Levenshtein distance [60] can be used to compare two strings for similarity, and it has been applied to the similar problem of finding duplicate web pages [65]. This algorithm, also known as string edit distance, takes two strings and gives a value to the distance between them by assigning costs to the actions needed to transform one of the strings into the other string.

The edit distance between the two strings *house* and *router* is 3 because the minimum number of edit operations needed to change the word *house* into *router* is three: substitute r for h, substitute t for s and insert r at the end.

Program source code is just a string, therefore the algorithm can be applied to two program source codes to determine how similar they are based on their syntactic make up.

2 Tokenisation

Tokenisation takes the string comparison techniques a bit further by pre-processing the source code before string comparison techniques are used. Tokenisation of a program involves scanning the source code and classifying sections of the code as tokens, each having a meaning. For example, the tokens in figure 3.2, page 66 could be a result of tokenising *int x = 5 * 3;*. Each token has a meaning along with the value of the original string. Tokenisation usually relies on regular expressions to match the sections of code to be turned into tokens.



TYPE	int
IDENTIFIER	x
ASSIGNMENT	=
NUMBER	5
MULTIPLY	*
NUMBER	3
SEMICOLON	;

Figure 3.2: Possible tokens for *int x = 5 * 3;*

A problem with string comparison, in terms of program source code comparison, is the large effect of, for example, changing identifier names while leaving the rest of the program the same as the original. This is where utilities such as *diff* cannot provide an accurate value of the similarity between two source code files. Tokenisation allows identifier names to be ignored by just replacing them with some marker in both programs being compared. Comparing the two pieces of code, *int x = 5 * 3;* and *int myVariable = 5 * 3;*, with string edit distance algorithms will result in a fairly large distance due to very different variable name used. But if both variable names, during tokenisation, where replaced by some other value in both programs (for example $), the programs would be matched exactly.

Prechelt *et al* [80] use tokenisation as part of their JPlag plagiarism detection system. They build on earlier work [104] by applying the greedy-string-tiling algorithm in their system to compare token strings. The success of JPlag is, in part, due to it's "complete ignorance of formatting, comments, literals and names" [80].

3 Software Metrics

One of the long used techniques in detecting source code plagiarism is software metrics. Software metrics are statistical values gained by parsing a program, such as how many operators or variables or methods etc there are in a program.

Halstead [45, 46] defined some basic software science parameters that have been used as a basis for metrics used in plagiarism detection in several systems [73, 30, 5, 54, 59] over the past 30 years. The basic software parameters, as defined by Halstead, are:

- n_1 - the number of unique operators
- n_2 - the number of unique operands
- N_1 - the total number of occurrences of operators
- N_2 - the total number of occurrences of operands

Halstead uses this to express the effort or complexity of a program as:

- $E = \frac{V}{L}$
- where $V = N log_2(n_1 + n_2)$
- and $N = n_1 log_2 n_1 + n_2 log_2 n_2$
- and $L = \frac{2}{n_1} + \frac{n_2}{N_2}$

There is no general consensus as to what constitutes an operator or operand in most programming languages therefore these need to be chosen depending on the language used, here we are only concerned with Java for which one source has suggested some guidelines [70].

Apart from Halstead's metrics we can consider attributes such as (and this list is not extensive):

- number of variables
- number of methods
- number of loops
- number of conditional statements
- number of method invocations

For example two programs with the same number of variables could be considered as suspects for plagiarism. Obviously two programs having the same number of variables does not necessarily mean they are plagiarised and this can only suggest they are looked at for further inspection. Choosing the correct metrics and the correct number of metrics is important in obtaining good results.

Though a better technique than string pattern matching metrics based systems still do not take into account the structure of a program and are only based on quantifiable statistics. Thus metric-based techniques are not always the best option for a plagiarism detection system [20].

4 Trees

A program is represented, in memory, as a parse tree or an AST. Two programs could be compared structurally by comparing the structure of their ASTs providing a more accurate comparison than metric or string based techniques. Some research has been been done in clone detection by comparing ASTs [7, 52, 84]. which is basically the same problem as plagiarism detection but for the purpose of finding duplicate code and improving re-use.

Figure 3.3 (page 68) shows the AST for program 3.1 (page 69) and figure 3.4 (page 68) shows the AST for program 3.2 (page 69). This programs are semantically equivalent with program 3.2 having an always true conditional statement. The two ASTs are very simliar with 3.4 having only two more nodes than 3.3.

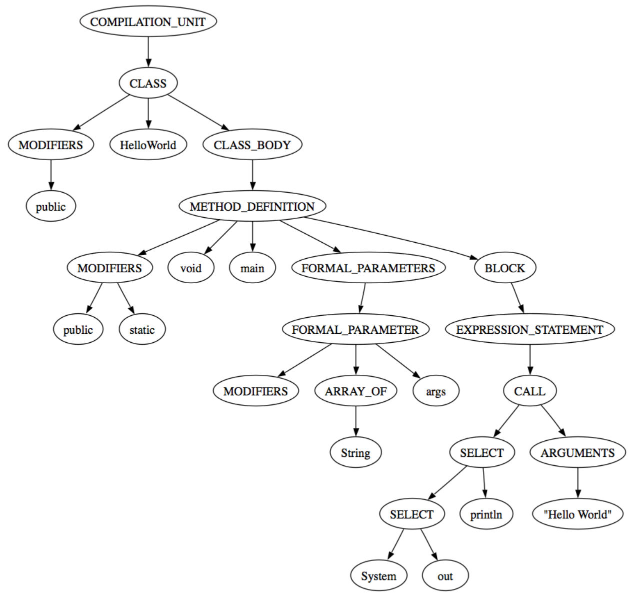


Figure 3.3: HelloWorld.java (Listing 3.1) AST

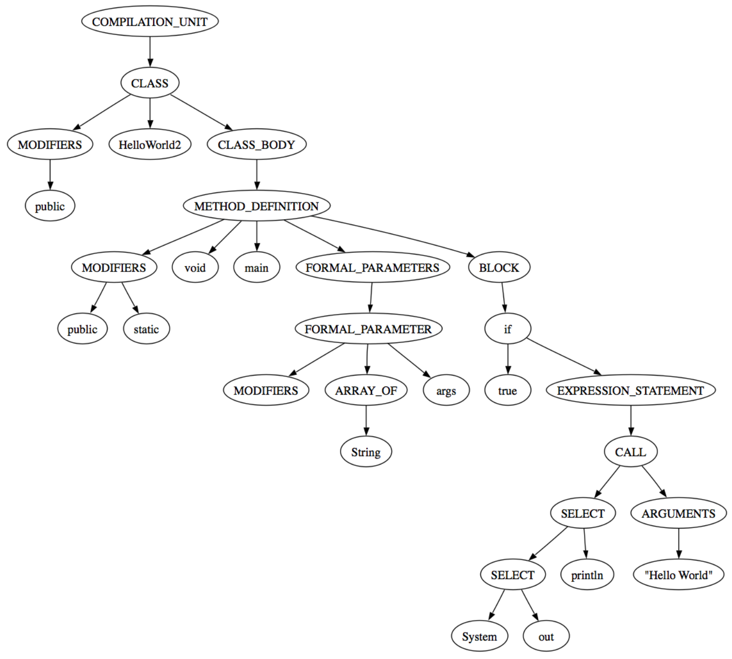


Figure 3.4: HelloWorld2.java (Listing 3.2) AST

Listing 3.1: HelloWorld.java

```java
public class HelloWorld {

        public static void main(String[] args) {
                System.out.println("Hello World");
        }

}
```

Listing 3.2: HelloWorld2.java

```java
public class HelloWorld2 {

        public static void main(String[] args) {
                if(true)
                        System.out.println("Hello World");
        }

}
```

The ASTs of programs can be compared with tree similarity algorithms of which there is much literature. For example, Sasha *et al.* [106] describe such algorithms in detail for exact and approximate matching and Valiente [100] provides some simpler algorithms and implementation examples. Sager *et al.* apply several of Valiente's algorithms to the problem of detecting similar Java classes in the field of clone detection but the same methods can be used for plagiarism detection, as it is, in essence, the same problem. Two methods involved finding the maximum number of common sub-trees, and another involved calculating the tree edit distance.

Finding common sub-trees shows how many parts of two trees are similar (Figure 3.5, page 69), whereas the tree edit distance between two trees is essentially the cost of the number of operations needed to transform one tree into the other: edither substitituion, deletion (Figure 3.6, page 70) or insertion. The tree edit distance is similar to the string edit distance.

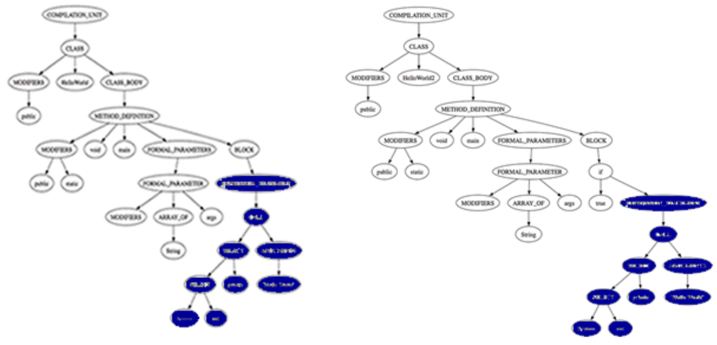


Figure 3.5: ASTs for programs 3.1 and 3.2 with two common sub-trees highlighted

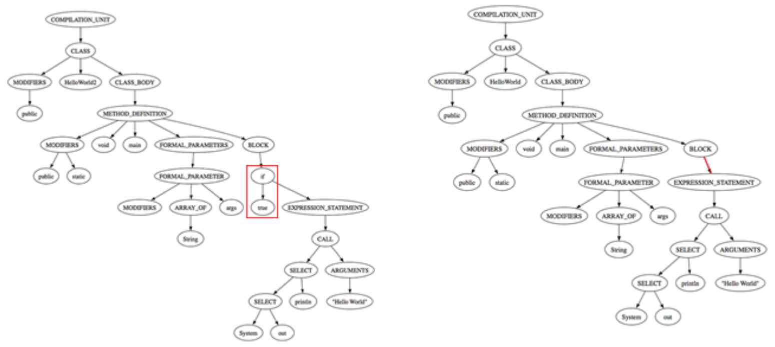


Figure 3.6: ASTs for programs 3.2 and 3.1 with nodes to remove highlighted

Jiang *et al.* [52] introduce a novel algorithm to reduce the cost of comparing a large number of sub-trees in large programs by differentiating between nodes which are relevant and those which are not.

Much of the research in the area of tree similarity for parse trees or ASTs has been in the area of clone detection not plagiarism detection. The two areas overlap somewhat and solving one problem essentially solves the other.

5 Call Graphs and Program-Dependency Graphs

A call hierarchy graph is a directed graph representing the method calls in a program from the methods that called them. A program dependency graph is a represention of the data flow and control dependencies in a program. Analysis and comparison of program dependency graphs has been applied to plagiarism detection [56, 62].

GPLAG is one such system which implements this technique with a good effectiveness compared to other tools [62]. Chilowicz *et al.* [17] applied call graph analysis to plagiarism detection in order to cluster similar functions between programs to detect plagiarism.

The programs can be parsed and call graphs or program dependancy graphs can be compared similarly to ASTs using a graph comparison algorithm [11] such as comparing the number of common sub-graphs or calculating the graph edit distance.

Figure 3.7, (page 71) shows a call graph of the simple program 3.7 (page 110) and figure 3.8 shows a call graph of the simple program 2 (page 111). Both programs are almost identical apart from an extra method call in program 2.

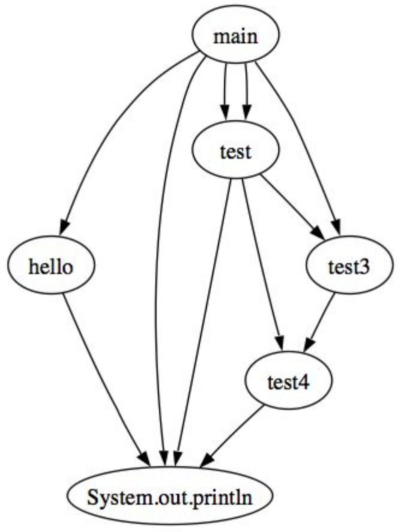


Figure 3.7: Call Graph of Listing 1

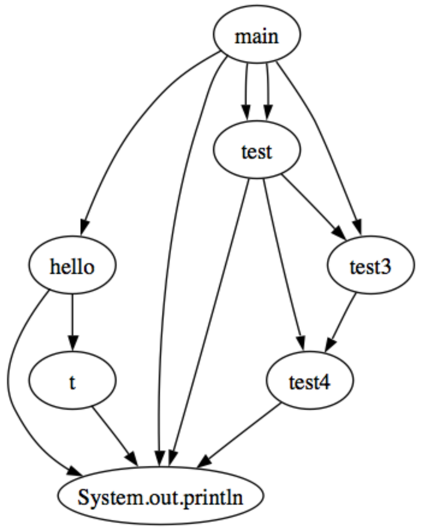
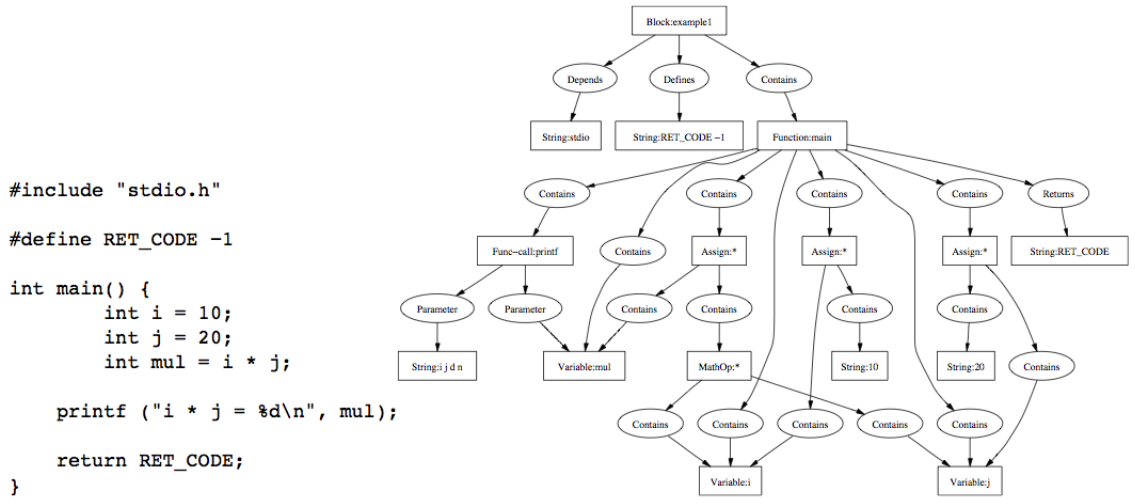


Figure 3.8: Call Graph of Listing 2

6 Conceptual Similarity

Mishne *et al.* [69] developed a source code retrieval system based on a programs conceptual similarity. Programs are represented in the form of an abstract conceptual graph where each node is either a *concept* or *relation* node (Figure 3.9). A graph made up of these two types of nodes represent an abstraction of the program which can be compared to find similarities. The results of this technique were, according to Mishne *et al.*, encouraging. The system, designed as a document retrieval system for source code files, is not specifically a plagiarism detection system but the techniques can be used as such.



```
#include "stdio.h"

#define RET_CODE -1

int main() {
        int i = 10;
        int j = 20;
        int mul = i * j;

    printf ("i * j = %d\n", mul);

    return RET_CODE;
}
```

Figure 3.9: example C program and it's conceptual graph. Source: [69]

7 Document Fingerprinting

One of the most popular systems used by academic institutions is MOSS [26]. MOSS uses a document fingerprinting technique; that is, it identifies some unique features in a document in order to give it a unique fingerprint. In theory every different document should have a different fingerprint; similar documents will have similar fingerprints.

Most document fingerprinting techniques work by using *k-grams*, where a *k-gram* is a contiguous substring of length *k*. Figure 3.10 (page 73) demonstrates an example of document fingerprinting: first remove whitespace characters, then derive all 5-grams from the text, apply some hash function to the grams, select a sub-set of these hashes using a function such as *0 mod 4*.

1. `public String s = "Hello";`

2. `publicStrings="Hello";`

3. `publi ublic blics licst icsta cstat stati tatic aticS ticSt icStr cStri Strin tring rings ings= ngs=" gs="H s="He ="Hel "Hell Hello ello" llo";`

4. `10701 11107 9382 10296 10003 9496 10975 11013 9314 11035 10000 9401 8022 11062 10851 10034 10476 9860 10805 5742 3364 6960 9660 10306`

5. `10701    10003    9314    8022    10476    3364`

Figure 3.10: Document fingerprinting example adapted from [86]

A disadvantage of the technique in figure 3.10 is that a *k-gram* shared by two documents is only found if it is in the sub-set selected in step 5. To overcome this problem Aiken *et al.* [86] developed a new technique known as Winnowing to select the sub-set of *k-grams*.

Winnowing involves creating 'windows' of the *k-gram* hashes (Figure 3.11, page 73), and select a subset as defined by:

> "In each window select the minimum hash value (Figure 3.12, page 73) [if it is not the same as the previous window]. If there is more than one hash with the minimum value, select the rightmost occurrence. Now save all select hashes as the fingerprints of the document [with positional information in original *k-grams* sequence] (Figure 3.13, 74)" [86]

```
[10701, 11107, 9382, 10296]     [11107, 9382, 10296, 10003]
[9382, 10296, 10003, 9496]      [10296, 10003, 9496, 10975]
[10003, 9496, 10975, 11013]     [9496, 10975, 11013, 9314]
[10975, 11013, 9314, 11035]     [11013, 9314, 11035, 10000]
[9314, 11035, 10000, 9401]      [11035, 10000, 9401, 8022]
[10000, 9401, 8022, 11062]      [9401, 8022, 11062, 10851]
[8022, 11062, 10851, 10034]     [11062, 10851, 10034, 10476]
[10851, 10034, 10476, 9860]     [10034, 10476, 9860, 10805]
[10476, 9860, 10805, 5742]      [9860, 10805, 5742, 3364]
[10805, 5742, 3364, 6960]       [5742, 3364, 6960, 9660]
[3364, 6960, 9660, 10306]
```

Figure 3.11: Windows of hashes of length 4 derived from 3.10

```
9382, 9496, 9314, 8022, 10034, 9860, 5742, 3364
```

Figure 3.12: Fingerprints selected by applying winnowing to the windows from figure 3.11

The algorithm is efficient due to the fact that the minimum value from the current window is, for most of the time, the minimum value in the previous window. Experiments showed that the winnowing algorithm is both efficient and guarantees that matches of a certain length are detected [86]. Source code must be pre-processed before using this algorithm, for example to remove identifier names which do not make a source file different.

```
[9382,2], [9496,5], [9314,8], [8022,12], [10034,15], [9860,18], [5742,20], [8364,21]
```

Figure 3.13: Fingerprints with 0-positional information from figure 3.10, step 4

8 Shared Information

"Two strings p and q can be said to be similar if the basic blocks of p are in q and vice versa" [42]

A compression algorithm will remove any redundent information from strings p and q by referencing the sub-sets of information from one in the other, thus $C(pq) < C(p) + C(q)$. The more information that is shared by p and q the greater the similarity between the two strings. This technique can be applied to any string including source code (though again it will be necessary to pre-process the source code to remove items such as identifier names).

Chen *et al.* [15] investigated the use of compression techniques and implemented SID[2] - Shared Information Distance or Software Integrity Detection. This system is based on the amount of shared information between strings and uses a compression algorithm, known as TokenCompress, to achieve this. Unlike MOSS full details of SID's implementation are available. Chen *et al.* [14] found that, for some source code, SID gives better results than MOSS or JPlag. AC also implements a compression based similarity algorithm [42].

[2]`http://genome.math.uwaterloo.ca/SID/`

9 Birthmarking

Program birth marking techniques use sections of code which are likely to be similar in a copied version of that code. A birthmark is created from a set of these sections which can then be compared with another birthmark to see how similar they are.

Birthmarking is related to watermarking where unique information which can be used to identify a piece of code is inserted into that code. The difference being that watermarks are inserted into a program whereas birthmarks are calculated from a program.

Tamada *et al.* [95] define 4 birthmarks for a plagiarism detection system which they implement for detecting similarity between Java class files.

Informally, these are (with examples for program 1, page 110):

- Constant Values in Field Variables - constants declared in one program may not be changed in the copied program as they might affect the output of the program (Figure 3.14, page 75)
- Sequence of Method Calls - list of method calls in the order they were called (Figure 3.15, page 75)
- Inheritance Structure - list of class hierarchies used in the program (Figure 3.16, page 75)
- Used Classes -list of all classes used in the program (Figure 3.17, page 75)

```
(int,7)
```

Figure 3.14: constant values in field variables birthmark for program 1

```
void hello(),void System.out.println(String),void test(),void test3(),void test(),
void System.out.println(String),void test3(),void test4(),void test4(),
void System.out.println(String),void System.out.println(String)
```

Figure 3.15: sequence of method calls birthmark for program 1

```
java.lang.Object
```

Figure 3.16: inheritance structure birthmark for program 1

```
java.lang.String,System
```

Figure 3.17: used classes birthmark for program 1

After computing birthmarks for a program the similarity to another birthmark can be achieved by calculating the percentage of elements occurring in both birthmarks.

10 Programming Style

Identifying the author of a program by the style with which it was written is a different approach from the other techniques presented here. Krsul *et al.* [57] investigate some techniques for analysing the authorship of a program using some software metrics such as where open/close curly brackets ({/}) occur within lines or preference of while or for loops. The experiments performed showed that within their closed environment the system performed well by correctly identifying authors of programs with very low error rates though they do suggest ideas for some new metrics for further work.

Chapter 4

Design and Implementation

In this chapter we design and implement a prototype plagiarism detector for a corpus of Java code source files with the aim of visualising the possible similarities between the programs and therefore expose the possibility of plagiarism. Displaying the data visually is important to aid a marker to quickly identify plagiarised assignments without cross checking every one manually, which is a non-trivial task for a human marker.

We implement the system in Java using the Eclipse JDT package which was chosen in the previous chapter as the tool most appropriate for this task.

$$D(P, t) = \{(p_i, p_j) : (p_i, p_j) \in P^2 \wedge S(p_i, p_j) \leq t\}$$

Figure 4.1: The input of the plagiarism detector D should be a set P of source files $\{p_1, p_2, \dots, p_n\}$ and a similarity threshold t. The output should be a sub-set of P^2 where the similarities of (p_i, p_j) are below the threshold t.

1 Visualisation

To provide a visualisation we look to Ellis *et al.* [35] who implemented a plagiarism detection system with a graphical representation of the similarity matrix (Figure 4.2, page 78) where each square represents a pair of programs and its shade represents their similarity. The darker the square is the more similar the two programs are. With just a quick glance at the visualisation a marker can easily see which pairs are similar.

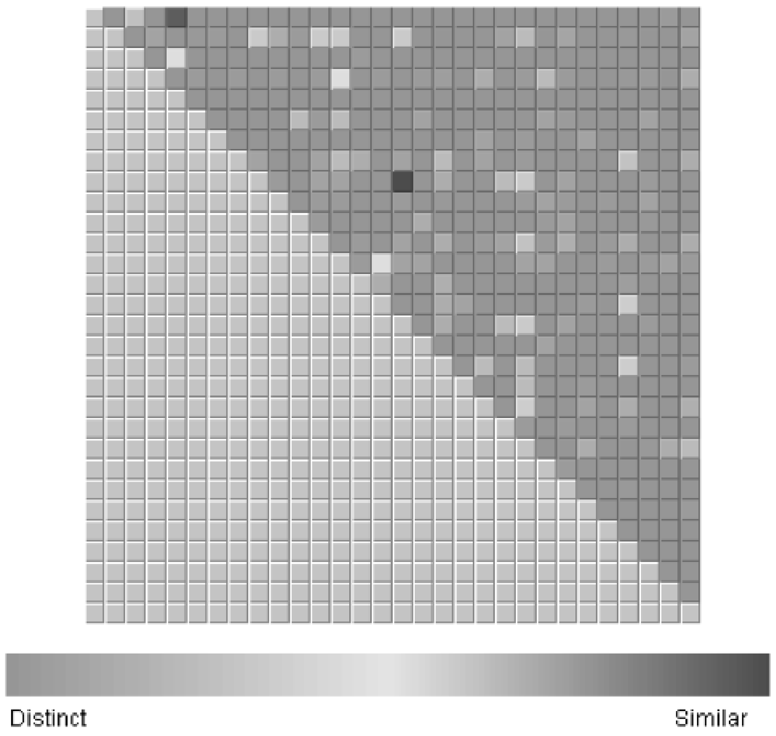


Figure 4.2: Similarity Matrix, source: [35]

2 Similarity Measures

We implement some similarity methods including some previously discussed metrics and a tree distance measure based on the the number of nodes at each level in the AST.

2.1 Document Length

We define document length of a Java program $L(p)$ as being the length of the string after the removal of any comments. Each document will most likely have a different length and though we do not expect this to provide a reliable metric it will be interesting to compare it to other metrics. It will result in a high similarity if only small changes are made to documents.

2.2 Variable Count

We define the variable count of a Java program $V(p)$ as the number of variables declared anywhere within the source file, for example either as field variables or method parameters; either static or not.

2.3 Method Count

We define the method count of a Java program $M(p)$ as being the number of methods declared anywhere within the source file, either static or not.

2.4 Method Invocation Count

We define the method invocation count of a Java program $C(p)$ as being the number of methods invoked anywhere within the source file, including static and non-static methods and constructors. Methods invoked can include locally declared methods or methods declared elsewhere (such as Java library calls).

2.5 Loop Count

We define the loop count of a Java program $W(p)$ as the number of loop constructs used anywhere within the source file, including *while*, *for* and *for-each*.

2.6 If Count

We define the if count of a Java program $I(p)$ as the number of *if* statements used anywhere with the source file.

2.7 Document Fingerprint

We define the document fingerprint of a Java program $F(p)$ as a set of integers returned by algorithm 1. For two programs we use the cardinality $F_c =\mid F(p_a) \cap F(p_b) \mid$ as the measure of similarity between the two fingerprint sets i.e. how many items exist in both sets. The closer F_c becomes to $\mid F(p_a p_b) \mid$ the more similar the files are.

Algorithm 1 Calculate Fingerprint Set

```
Require: program $p$, integer $k$ denoting gram size
  $S \leftarrow new$ set
  $remove\_comments(p)$
  $remove\_whitespace(p)$
  for $i = 0$ to $length(p) - k$ do
    if $i \bmod 4 = 0$ then
      $s \leftarrow substring(p, i, i + k)$
      add $hash(s)$ to $S$
    end if
  end for
  return $S$
```

2.8 AST Node Count

We define AST node count of a Java program $A(p)$ as a vector $(d_1, d_2, \ldots, d_n)$ representing the number of nodes at each depth in a Java programs AST. We then compute the similarity between two programs p_a and p_b as the Euclidean distance between the two vectors:

$$A_d = d(x, y) = \sqrt{\sum_{k=1}^{n} (x_i - y_i)^2}$$

where

$$x = A(p_a), y = A(p_b)$$

The smaller the distance A_d the more similar programs p_a and p_b are.

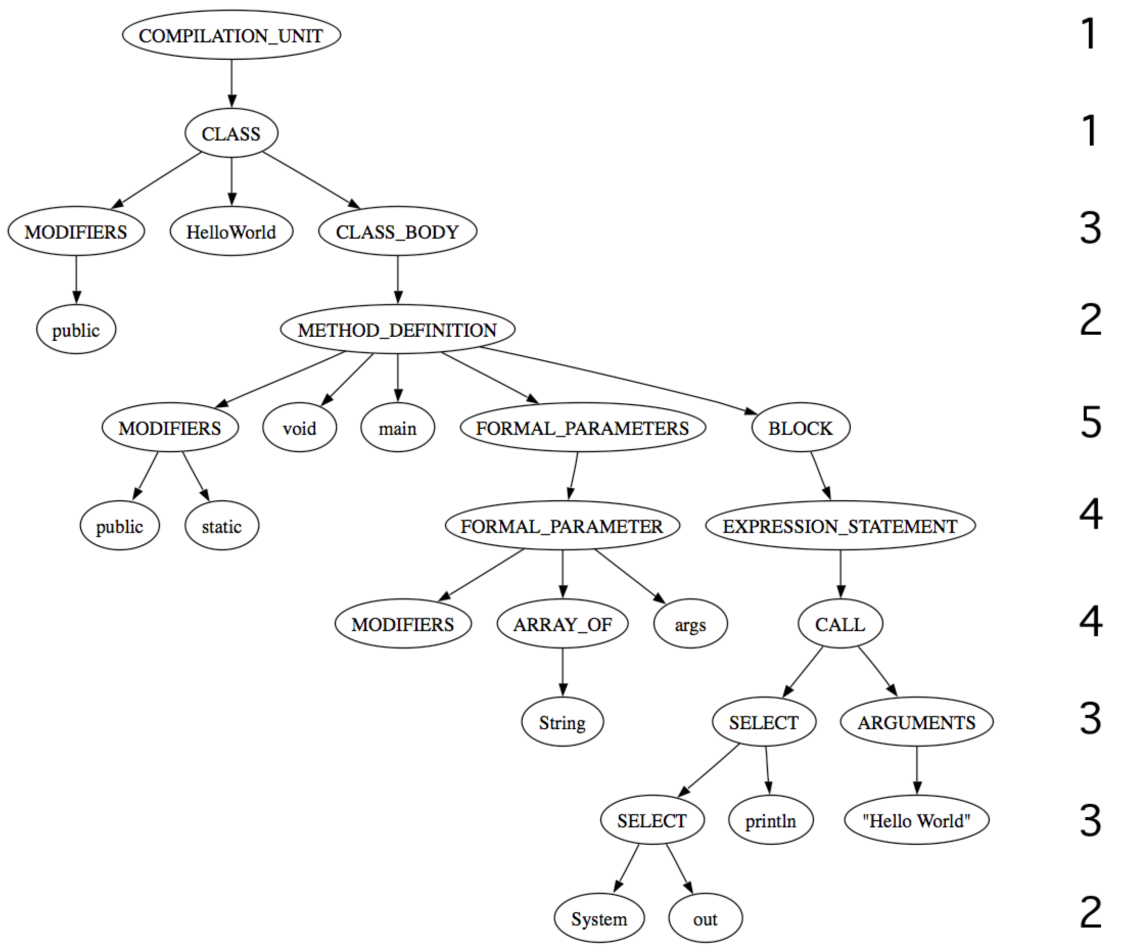


Figure 4.3: AST node depths for program 2.46, page 55

3 Distance

In order to quantify the similarity between two programs we calculate the distance in Euclidean n-*dimensional* space E^n between vectors whose co-ordinates are composed of the results of the similarity measures. In the case of using all our similarities measures we have 8 dimensions but we could decide to not include all of them or we could add more measures in the future. We can define a vector of a program as

$$V_p = (L(p), V(p), M(p), C(p), W(p), I(p), F_c(p), A_d(p))$$

and the distance between two program vectors V_a and V_b as

$$D_{ab} = d(x, y) = \sqrt{\sum_{k=1}^{n} (x_i - y_i)^2}$$

where

$$x = V_a, y = V_b$$

We compute a distance matrix D from a set of programs P where each D_{ab} represents the distance between two programs P_a and P_b then normalise the values into the range $0 \ldots 100$

$$D_{ab} = 100 - \frac{100}{max(D) \times D_{ab}}$$

where 0 is most similar and 100 is most different.

A threshold value t will allow us to ignore very different programs and concentrate on the most similar though it is not yet known what would be a good value for t.

4 Implementation

We implemented the system in Java using Object Orientated design principles[1]. Class diagrams for the system show each package and the classes contained within (Figure 4.4, page 83; 4.5, page 84; 4.6, page 85).

The application uses a Java Swing interface along with the Java2D API to provide a visualisation of the results (Figure 4.7, page 86). Each square in the grid represents a pair of programs and it's colour the measure of similarity between those two programs. The colour ranges from white meaning exact match (as demonstrated by the leading diagonal) to red meaning most different among the set.

A multi-select list box allows the selection of measures to be used to compare the programs the vector V_p for program is derived from the selected measures.

A slider is provided to set the threshold value t which allows the user to ignore very different program pairs. In figure 4.7, page 86 the threshold is at 0 meaning that all the program pairs are shown on the diagram. Figure 4.8, page 86 shows the same set of programs with a threshold of 75 meaning that only program pairs which meet the similarity value of 75 or greater are shown. In both examples the only similarity measure used is the AST measure A_d.

Figure 4.9, page 87 shows the use of all similarity measures on the same set of programs. Notice that we had to increase the threshold to get approximately the same program pairs as in figure 4.8.

[1]Full source code is available from `http://whoyouknow.co.uk/uni/msci/`.

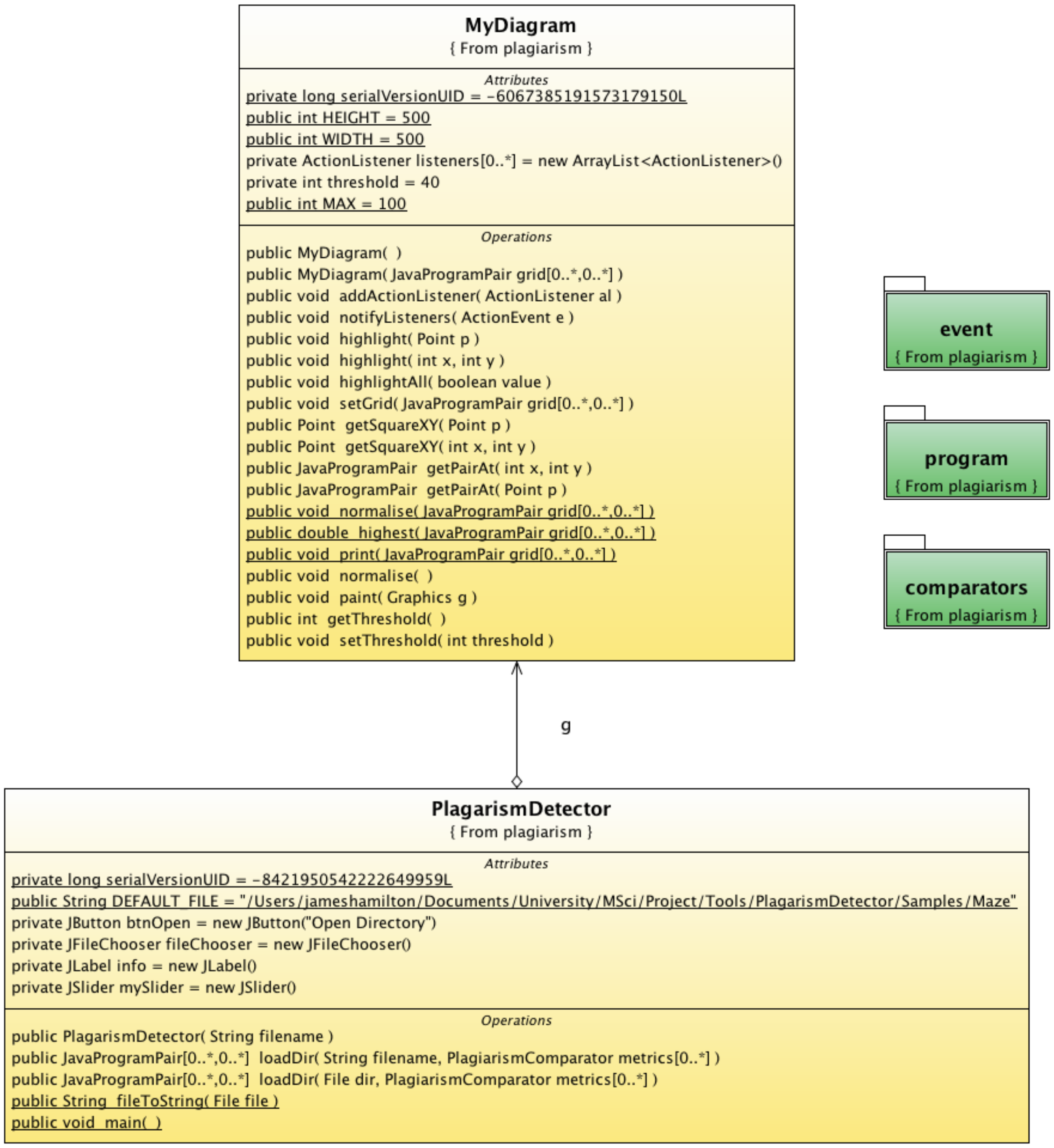

MyDiagram
{ From plagiarism }
Attributes
private long serialVersionUID = -6067385191573179150L
public int HEIGHT = 500
public int WIDTH = 500
private ActionListener listeners[0..*] = new ArrayList<ActionListener>()
private int threshold = 40
public int MAX = 100
Operations
public MyDiagram( )
public MyDiagram(JavaProgramPair grid[0..*,0..*])
public void addActionListener(ActionListener al)
public void notifyListeners(ActionEvent e)
public void highlight(Point p)
public void highlight(int x, int y)
public void highlightAll(boolean value)
public void setGrid(JavaProgramPair grid[0..*,0..*])
public Point getSquareXY(Point p)
public Point getSquareXY(int x, int y)
public JavaProgramPair getPairAt(int x, int y)
public JavaProgramPair getPairAt(Point p)
public void normalise(JavaProgramPair grid[0..*,0..*])
public double highest(JavaProgramPair grid[0..*,0..*])
public void print(JavaProgramPair grid[0..*,0..*])
public void normalise( )
public void paint(Graphics g)
public int getThreshold( )
public void setThreshold(int threshold)
event
{ From plagiarism }
program
{ From plagiarism }
comparators
{ From plagiarism }
g
PlagarismDetector
{ From plagiarism }
Attributes
private long serialVersionUID = -8421950542222649959L
public String DEFAULT_FILE = "/Users/jameshamilton/Documents/University/MSci/Project/Tools/PlagarismDetector/Samples/Maze"
private JButton btnOpen = new JButton("Open Directory")
private JFileChooser fileChooser = new JFileChooser()
private JLabel info = new JLabel()
private JSlider mySlider = new JSlider()
Operations
public PlagarismDetector(String filename)
public JavaProgramPair[0..*,0..*] loadDir(String filename, PlagiarismComparator metrics[0..*])
public JavaProgramPair[0..*,0..*] loadDir(File dir, PlagiarismComparator metrics[0..*])
public String fileToString(File file)
public void main( )

Figure 4.4: Plagiarism Package Class Diagram

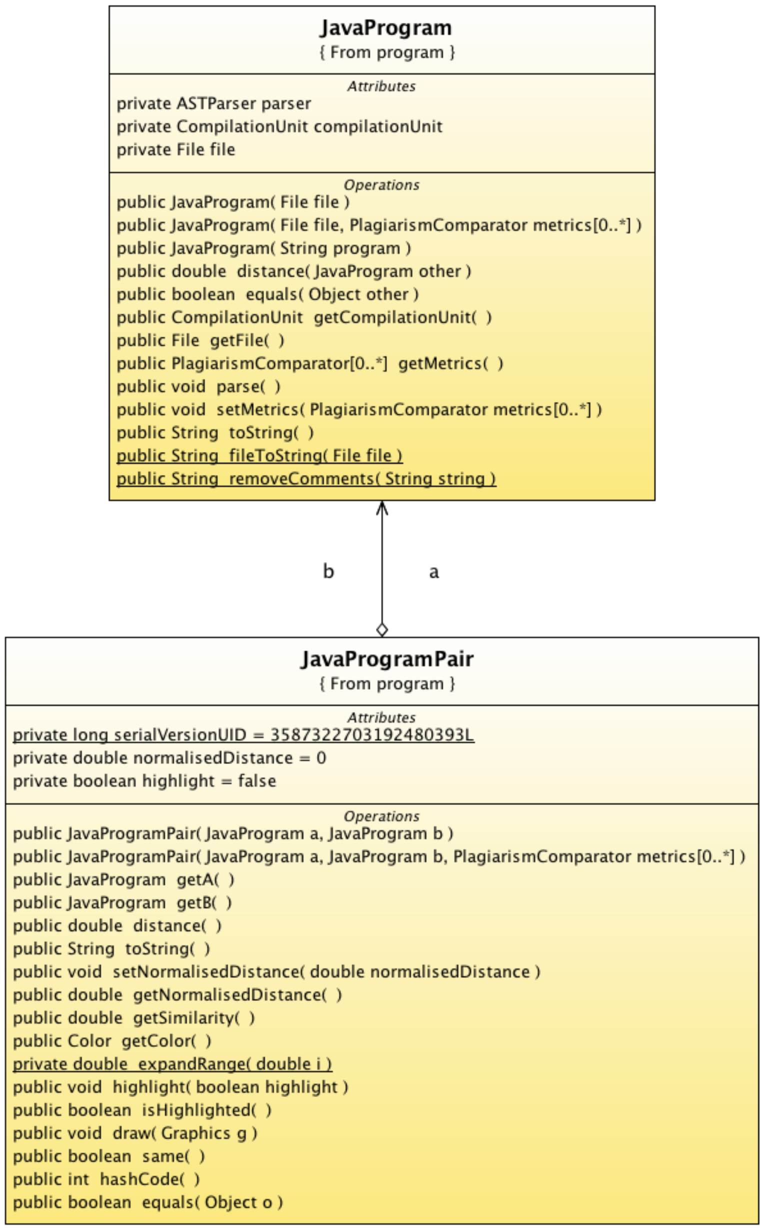

JavaProgram
{ From program }
Attributes
private ASTParser parser
private CompilationUnit compilationUnit
private File file
Operations
public JavaProgram(File file)
public JavaProgram(File file, PlagiarismComparator metrics[0..*])
public JavaProgram(String program)
public double distance(JavaProgram other)
public boolean equals(Object other)
public CompilationUnit getCompilationUnit( )
public File getFile( )
public PlagiarismComparator[0..*] getMetrics( )
public void parse( )
public void setMetrics(PlagiarismComparator metrics[0..*])
public String toString( )
public String fileToString(File file)
public String removeComments(String string)
b
a
JavaProgramPair
{ From program }
Attributes
private long serialVersionUID = 3587322703192480393L
private double normalisedDistance = 0
private boolean highlight = false
Operations
public JavaProgramPair(JavaProgram a, JavaProgram b)
public JavaProgramPair(JavaProgram a, JavaProgram b, PlagiarismComparator metrics[0..*])
public JavaProgram getA( )
public JavaProgram getB( )
public double distance( )
public String toString( )
public void setNormalisedDistance(double normalisedDistance)
public double getNormalisedDistance( )
public double getSimilarity( )
public Color getColor( )
private double expandRange(double i)
public void highlight(boolean highlight)
public boolean isHighlighted( )
public void draw(Graphics g)
public boolean same( )
public int hashCode( )
public boolean equals(Object o)

Figure 4.5: Program Package Class Diagram

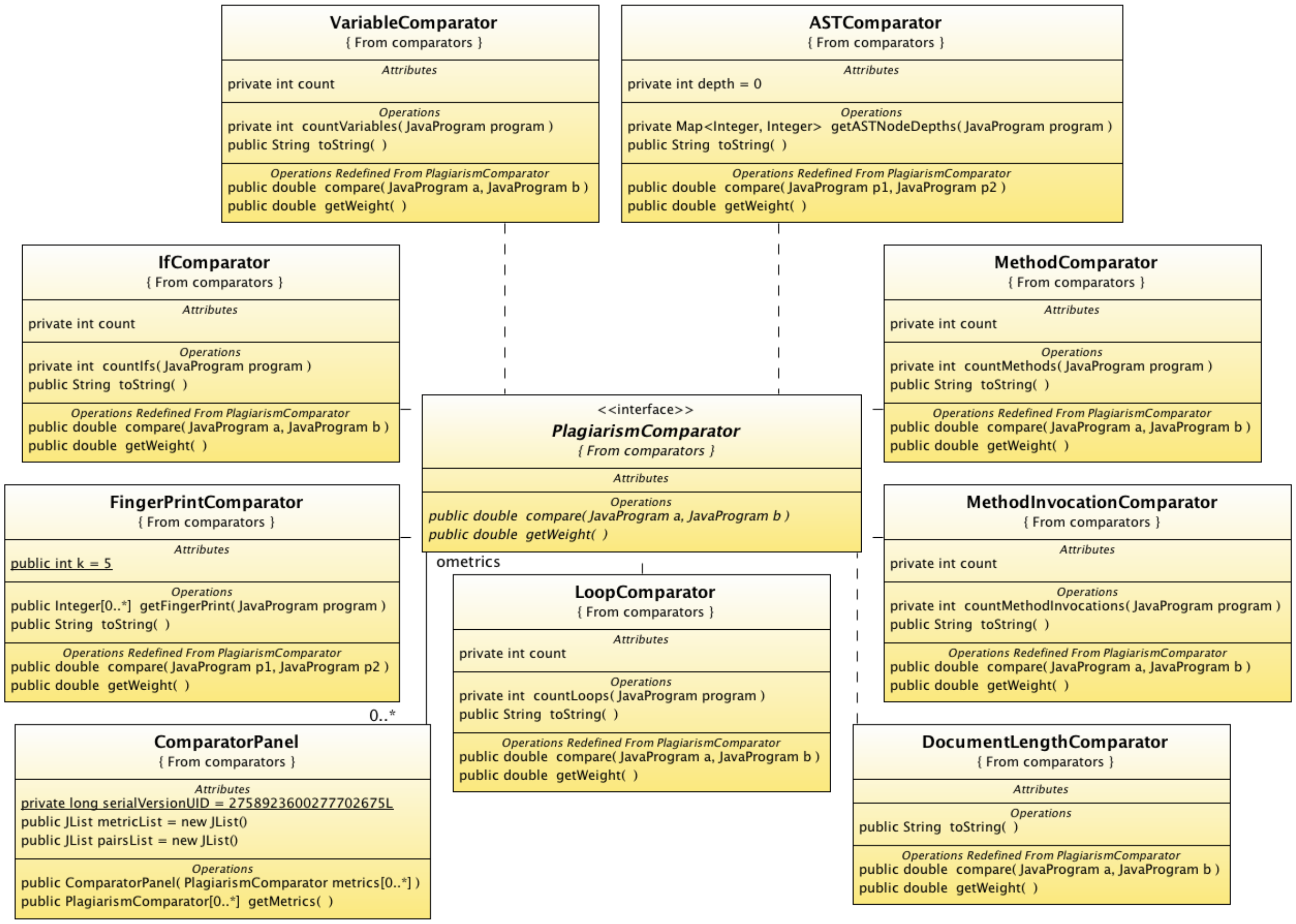

VariableComparator
{ From comparators }
Attributes
private int count
Operations
private int countVariables(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
ASTComparator
{ From comparators }
Attributes
private int depth = 0
Operations
private Map<Integer, Integer> getASTNodeDepths(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram p1, JavaProgram p2)
public double getWeight( )
IfComparator
{ From comparators }
Attributes
private int count
Operations
private int countIfs(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
<<interface>>
PlagiarismComparator
{ From comparators }
Attributes
Operations
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
MethodComparator
{ From comparators }
Attributes
private int count
Operations
private int countMethods(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
FingerPrintComparator
{ From comparators }
Attributes
public int k = 5
Operations
public Integer[0..*] getFingerPrint(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram p1, JavaProgram p2)
public double getWeight( )
ometrics
LoopComparator
{ From comparators }
Attributes
private int count
Operations
private int countLoops(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
MethodInvocationComparator
{ From comparators }
Attributes
private int count
Operations
private int countMethodInvocations(JavaProgram program)
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )
0..*
ComparatorPanel
{ From comparators }
Attributes
private long serialVersionUID = 2758923600277702675L
public JList metricList = new JList()
public JList pairsList = new JList()
Operations
public ComparatorPanel(PlagiarismComparator metrics[0..*])
public PlagiarismComparator[0..*] getMetrics( )
DocumentLengthComparator
{ From comparators }
Attributes
Operations
public String toString( )
Operations Redefined From PlagiarismComparator
public double compare(JavaProgram a, JavaProgram b)
public double getWeight( )

Figure 4.6: Comparators Package Class Diagram

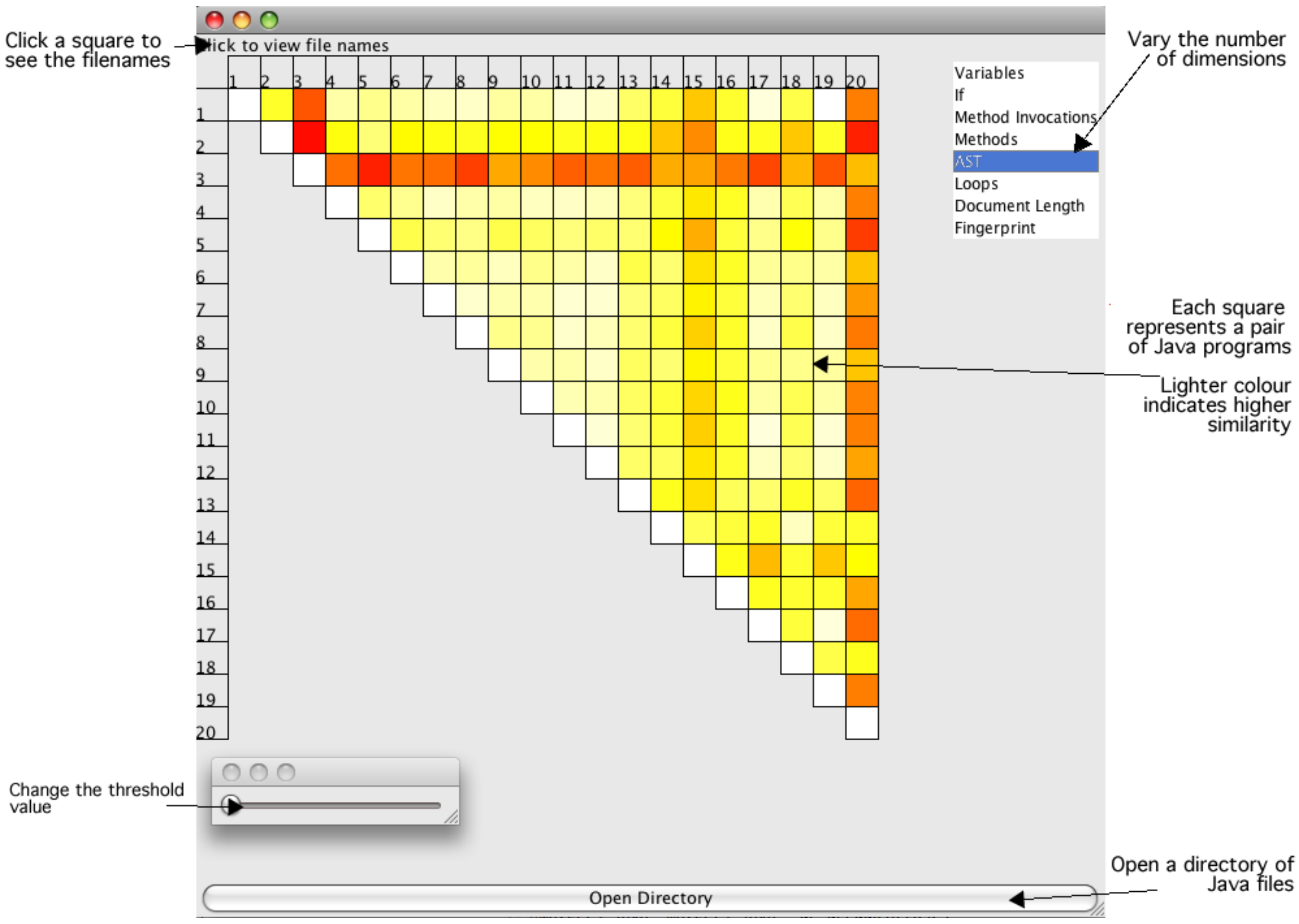

Click a square to see the filenames
Click to view file names
Vary the number of dimensions
Variables
If
Method Invocations
Methods
AST
Loops
Document Length
Fingerprint
Each square represents a pair of Java programs
Lighter colour indicates higher similarity
Change the threshold value
Open a directory of Java files
Open Directory

Figure 4.7: System screenshot

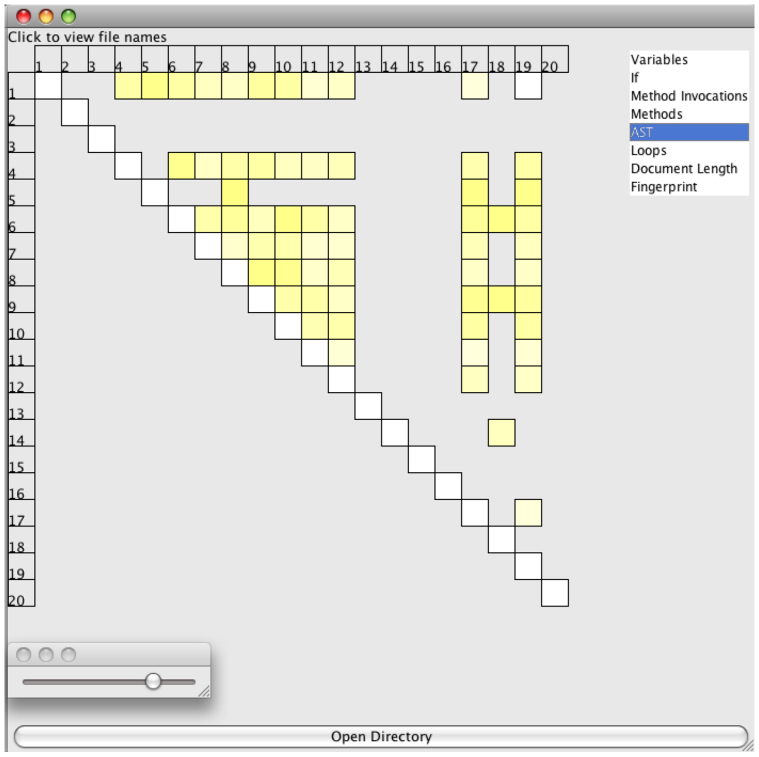


Figure 4.8: System screenshot with a higher threshold than in figure 4.7

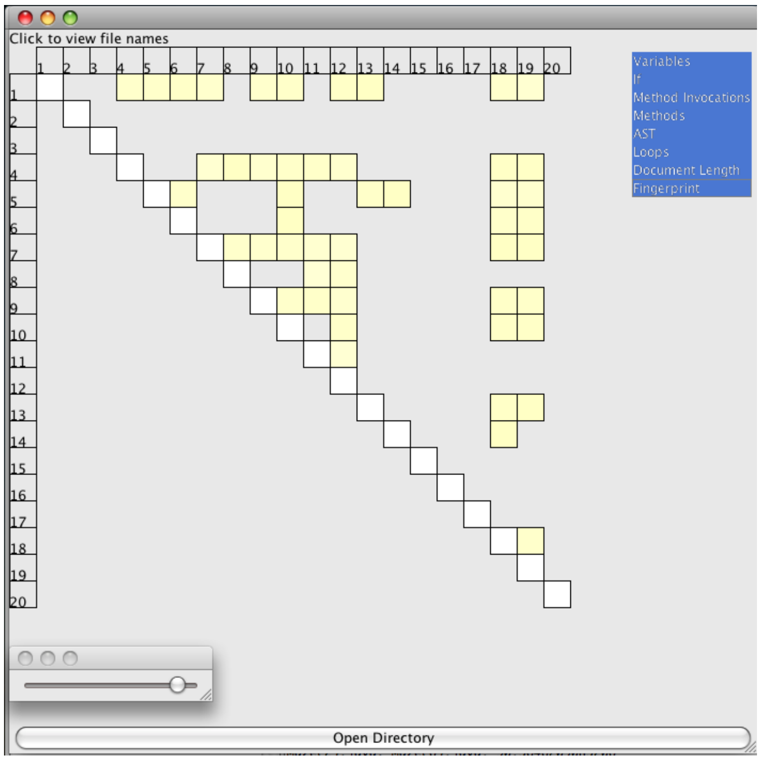


Figure 4.9: System screenshot with more similarity measures used and a higher threshold than in figure 4.8

5 Conclusion

It is unknown at this stage which similarity measures are the best to use but it is likely that we will need to eliminate some to produce a more accurate output. It is also unknown what a good threshold will be though we do know that different similarity measures will require a different threshold (as demonstrated by figures 4.8 and 4.9.

Different ranges produced by the measures e.g. document length $D(p)$ will produce a large figure may skew the results, while also not providing a good metric. We therefore hypothesise that we will need to choose a sub-set of measures for V_p in order to provide the best similarity measure. We also believe that document length will not be a very useful similarity measure and the most useful will be the AST measure as it takes into account program structure. We undertake an empirical study in the next chapter, using a corpus of real data, to test these hypotheses.

Chapter 5

Empirical Study

In order to select the best sub-set of measures for V_p we undertake an empirical study using a corpus of files from a first year Java programming course at Goldsmiths, University of London. It is, as yet, unknown which measures will provide the best similarity measure but we do not believe document length $L(p)$ will be included in the final measures as it doesn't take into account anything other than the length of the document. We believe the AST node count measure A_d will provide a better measure as it takes into account the structure of a program. This chapter describes our empirical study and discuses the results.

1 Assignments

The following are assignments taken from the course study guide [27] for which we have between 13 and 30 student submissions:

Rolling a Die "Write a program which emulates rolling a die. Every time the program is run, it outputs a random number between 1 and 6." 30 submissions.

Leap Years "Write a program in which the user enters a year and the program says whether it is a leap year or not." 28 submissions.

Drawing a Square "Using lineTo and moveTo, from element.jar, write a program that asks the user to enter an integer size which draws a square of that size." 25 submissions.

How Old Are You? "Write a program which asks the user for their date of birth and then tells them how old they are." 25 submissions.

Guessing Game "Write a program that tries to guess the number thought of by the user. The number is between 0 and 1000. If the computers guess is too high, the user should enter 2. If the computers guess is too low, the user should enter 1. If the computers guess is correct, the user should enter any integer except 1 or 2. Print out how many guesses it took the computer. Also print out if the user cheated!" 26 submissions.

Maze "Write a Java program that represents a maze. The maze must have a start and a nish. The idea is to move the mouse from the start to the nish without going outside the maze. The program should give an error message if the mouse goes off the path of the maze and force the user to start again by ending the program. If the user gets from the start to the nish successfully the program should display to the user how long it took in seconds." 19 submissions.

Hangman "The computer thinks of a word. (In fact, hard-wire the word into your program.) The user tries to guess the word by trying a letter at a time. If the letter is in the word, then the computer shows the user where it ts. Carry on in this way until either the user runs out of goes (say 9) or the user guesses the word." 13 submissions.

Roman Numerals "Romans used a strange way of representing numbers which we call roman numerals. In roman numeral notation, M stands for 1000, D for 500, C for 100, L for 50, X for 10, V for 5 and I for one. In order to compute the integer value of a roman numeral, you rst look for consecutive characters where the value of the rst is less than the second. Take, for example

XC and CM. In these cases you subtract the rst from the second, for example XC is 90 and CM is 900. Having done that, you simply add up all the values of such pairs and then to this add the values of the remaining individual roman numerals so, for example, MMMCDXLIX is 3449 and MCMXCIX is 1999.” 18 submissions.

2 Finding the Best Similarity Measures

In order to decide on the best similarity measures to include in our measure V_p we chose, at random, the *maze* corpus $C = \{p_i : 1 \leq i \leq 19\}$ and began our testing by manually checking for similarities between programs. We believe we have found 16 pairs of programs which seem highly likely to be plagiarised and we call this set $A = \{(p_a, p_b)_i : 1 \leq i \leq 16 \wedge (p_a, p_b)_i \in C^2\}$

In chapter 4 we described our set of 8 similarity measures M of which there are 256 different combinations in the powerset $\mathcal{P}(M)$. We therefore modify our program to automatically generate $\mathcal{P}(M)$ and apply the family of sets over M to all pairs in C^2.

Applying the measures in every sub-set of $\mathcal{P}(M)$ to every pair in C^2 allows us to see the result of trying every combination of measures on the pairs in C^2 and gives us a set B_i for set of measures i.

We then order the pairs in based on their similarity to see how many of the 16 known plagiarised pairs exist in the top 16 so the closer $|A \cap B_i|$ is to $|A|$ the better the choice of measures i is.

2.1 Results

We found that no combination of the 8 similarity measures found all 16 program pairs that we have decided may be plagiarised but 32 combinations put 8 of the 16 in the top ten. Interestingly the AST node count measure A_d on its own is one of these and it also appears in the 31 other combinations.

The following list shows all the possible values of V_p which are best for program similarity comparison based on the maze set C:

1. (A_d)
2. (A_d, W)
3. (M, A_d)
4. (M, A_d, W)
5. (C, A_d)
6. (C, A_d, W)
7. (C, M, A_d)
8. (C, M, A_d, W)
9. (I, A_d)
10. (I, A_d, W)
11. (I, M, A_d)
12. (I, M, A_d, W)
13. (I, C, A_d)
14. (I, C, A_d, W)
15. (I, C, M, A_d)
16. (I, C, M, A_d, W)
17. (V, A_d)
18. (V, A_d, W)
19. (V, M, A_d)
20. (V, M, A_d, W)
21. (V, C, A_d)
22. (V, C, A_d, W)
23. (V, C, M, A_d)

24. (V, C, M, A_d, W)
25. (V, I, A_d)
26. (V, I, A_d, W)
27. (V, I, M, A_d)
28. (V, I, M, A_d, W)
29. (V, I, C, A_d)
30. (V, I, C, A_d, W)
31. (V, I, C, M, A_d)

It is not really surprising that A_d appears in all of the combinations of measures because it takes into account program structure.

Another result that we are not surprised about is the accuracy of the document length L measure which only takes into account the length of the string representing the document. Many of the combinations lower in the list contained the document length measure.

Later we will try out a sub-set of these combinations on our other Java corpuses to see if we can find plagiarised pairs within then. It is most likely that we could just use the AST node count A_d metric on it's own to provide a good similarity as, in the maze corpus, it was the only measure which worked without others. This is due to it comparing the structure of a program and in doing so it actually takes into account attributes such as number of variables.

3 Finding the Best Similarity Threshold

So far we have not thought about the threshold value t. This value will allow the marker to ignore very different program pairs by provided a minimum similarity value for which the set of pairs is chosen. Previous we have found the best 31 combinations of similarity measures and now use those to found out what the best threshold value t to use with them is.

We have found 8 plagiarised programs using 31 combinations of similarity measures so we want to find $max(t)$ for $|A \cap B_i| = 8$ i.e. what is the highest value that t can be so that the number of found plagiarised pairs using set of measures i is 8.

We modify our program so that $t = 0$ and is incremented by one each time until we find that $|A \cap B_i| = 8$ no longer holds.

3.1 Results

All combinations of the similarity measures required threshold of 13 in order to keep the 8 program pairs listed.

The fact that all combinations require the same threshold has made us realise that probably the high value of A_d is overpowering the others measures and A_d is really the only measure taken into count. But we believe that the AST measure is the best compared to the others as it takes into account the program structure.

Now we believe that AST node count measure is the best for the maze set we can try this theory on the remaining assignment sets.

4 Testing the Measures

We will test the AST measure A_d on the other assignment corpuses to see if we can find plagiarised pairs with a threshold value of 13[1].

4.1 Maze

Figure 5.8, page 98 shows the visualisation of the maze assignments. This includes 8 out of the 16 pairs that we have determined to be plagiarised manually and some false positives.

Interestingly three pairs of programs which we noted down as plagiarised in the first instance and subsequently decided they were not plagiarised also appear in the results. We may have been wrong with deciding these pairs are not plagiarised after re-inspecting them after realising they were in the results of our plagiarism detector - it seems as though we may have been correct in our original decision as the first half of the programs are very similar even though the second halfs are different (Figure 5.2, page 95).

We also discovered another plagiarised pair (P_3, P_{10}) that we missed during our manual check (Figure 5.3, page 95). Pairs (P_3, P_{10}) and (P_6, P_{11}) are also similar.

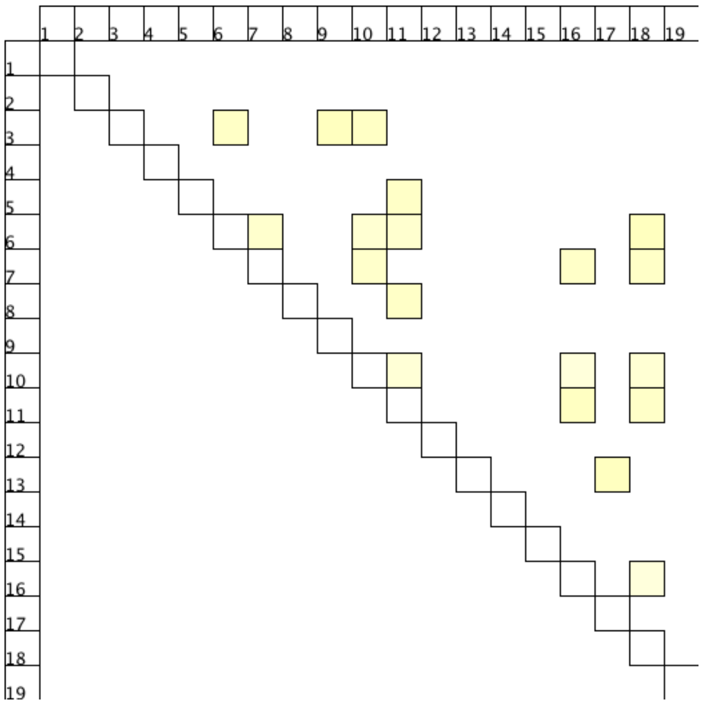
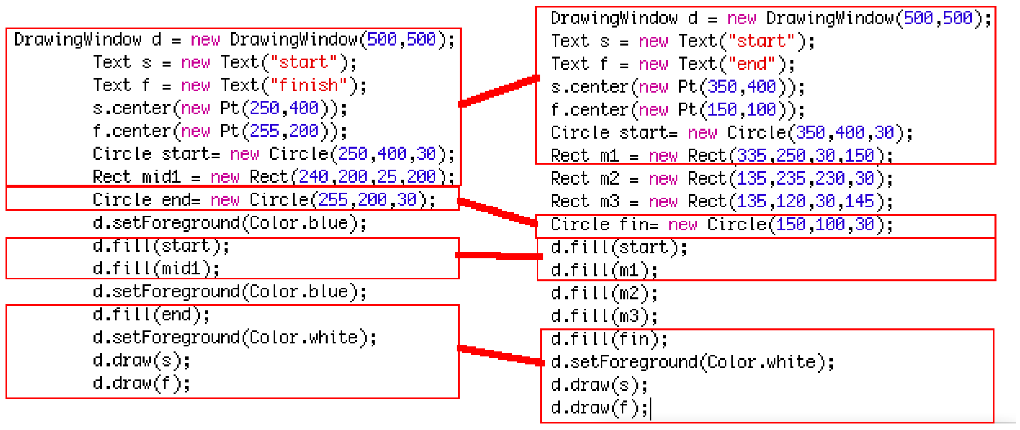


Figure 5.1: Maze with A_d measure and $t = 13$

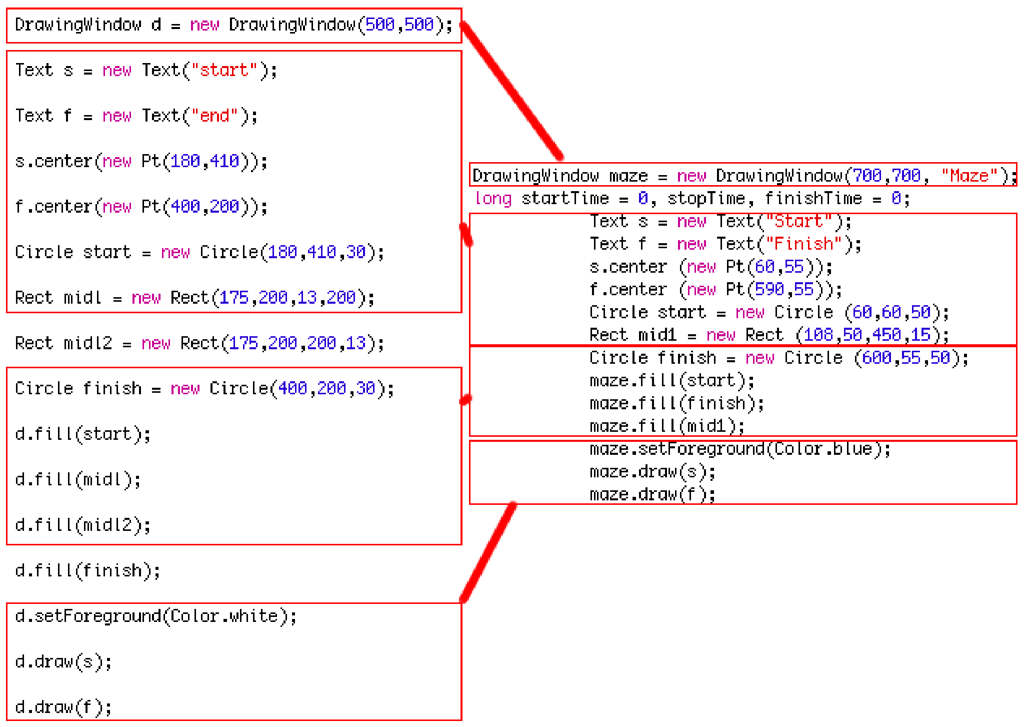
[1]Please note that from the diagrams it is sometimes not obvious if a square is white or blank (i.e. 0 score or above threshold) - this is something that we intend to fix.



```
DrawingWindow d = new DrawingWindow(500,500);
        Text s = new Text("start");
        Text f = new Text("finish");
        s.center(new Pt(250,400));
        f.center(new Pt(255,200));
        Circle start= new Circle(250,400,30);
        Rect mid1 = new Rect(240,200,25,200);
        Circle end= new Circle(255,200,30);
        d.setForeground(Color.blue);
        d.fill(start);
        d.fill(mid1);
        d.setForeground(Color.blue);
        d.fill(end);
        d.setForeground(Color.white);
        d.draw(s);
        d.draw(f);
```

```
DrawingWindow d = new DrawingWindow(500,500);
Text s = new Text("start");
Text f = new Text("end");
s.center(new Pt(350,400));
f.center(new Pt(150,100));
Circle start= new Circle(350,400,30);
Rect m1 = new Rect(335,250,30,150);
Rect m2 = new Rect(135,235,230,30);
Rect m3 = new Rect(135,120,30,145);
Circle fin= new Circle(150,100,30);
d.fill(start);
d.fill(m1);
d.fill(m2);
d.fill(m3);
d.fill(fin);
d.setForeground(Color.white);
d.draw(s);
d.draw(f);
```

Figure 5.2: Extract from programs P_6 and P_{11}



```
DrawingWindow d = new DrawingWindow(500,500);

Text s = new Text("start");

Text f = new Text("end");

s.center(new Pt(180,410));

f.center(new Pt(400,200));

Circle start = new Circle(180,410,30);

Rect midl = new Rect(175,200,13,200);

Rect midl2 = new Rect(175,200,200,13);

Circle finish = new Circle(400,200,30);

d.fill(start);

d.fill(midl);

d.fill(midl2);

d.fill(finish);

d.setForeground(Color.white);

d.draw(s);

d.draw(f);
```

```
DrawingWindow maze = new DrawingWindow(700,700, "Maze");
long startTime = 0, stopTime, finishTime = 0;
            Text s = new Text("Start");
            Text f = new Text("Finish");
            s.center (new Pt(60,55));
            f.center (new Pt(590,55));
            Circle start = new Circle (60,60,50);
            Rect mid1 = new Rect (108,50,450,15);
            Circle finish = new Circle (600,55,50);
            maze.fill(start);
            maze.fill(finish);
            maze.fill(mid1);
            maze.setForeground(Color.blue);
            maze.draw(s);
            maze.draw(f);
```

Figure 5.3: Extract from programs P_3 and P_{10}

4.2 Rolling a Die

Most programs in the Die set are very similar. It is impossible to say if people have plagiarised due to the simplicity of the task. Figure 5.4, page 96 shows nearly all programs fall within the threshold value, those that do not are students who went beyond the requirements of the assignment such as displaying a die graphically.

This most interesting thing we can say about this set is that we can infer which students are better at programming than others by looking at the diagram as most of the pairs are close to 0 and a few are above 80. Of course the case may actually be that the individual is a bad programmer and has written a two line program in thirty lines.

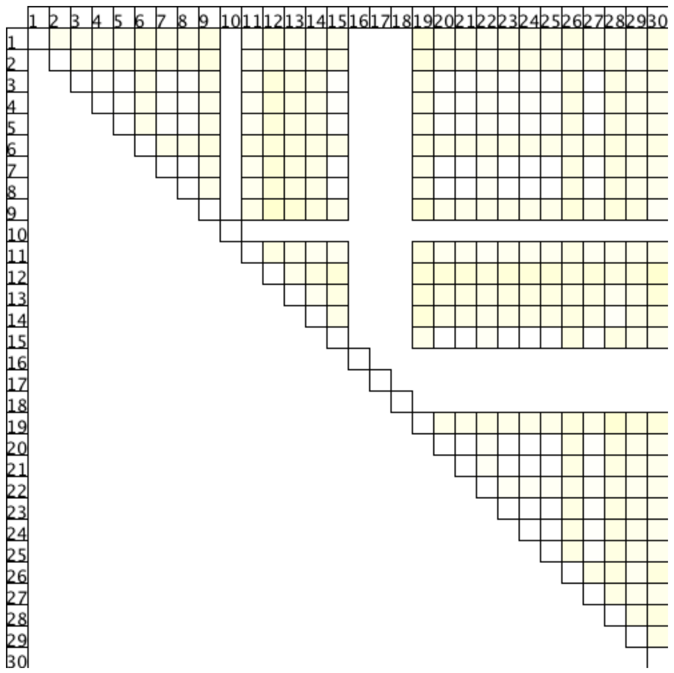


Figure 5.4: Rolling a Die with A_d measure and $t = 13$

4.3 Leap Year

In the leap year set there are 160 pairs which appear below the threshold. We can tell with a quick glance that some squares are white meaning they are exactly matching or at least they appear to be (we can say for sure using the AST node count measure as two different ASTs can have the same number of nodes at each depth). Figure 5.6, page 97 shows the plagiarised pair (P_8, P_{14}).

A pair with a similarity value of 1.8 is (P_{17}, P_{27}) which can be seen in Figure 5.7, page 98. This pair differs in that one has a declaration and assignment as one statement where the other has divided it into two[2].

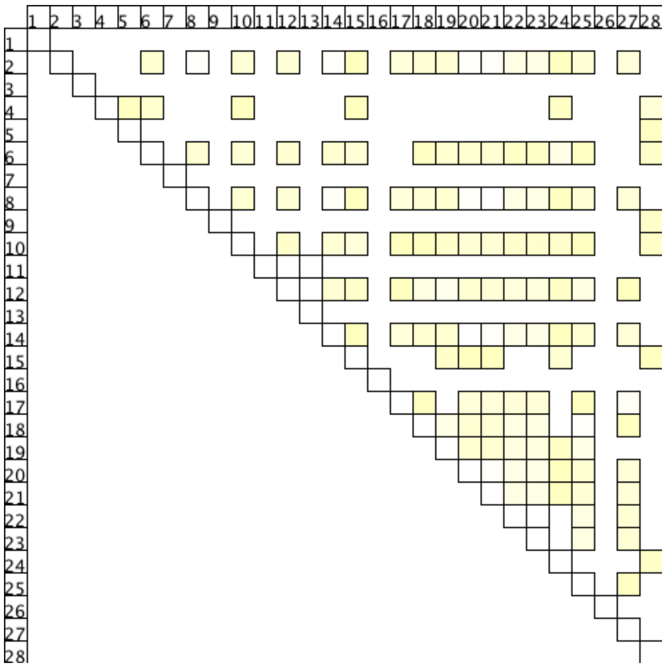
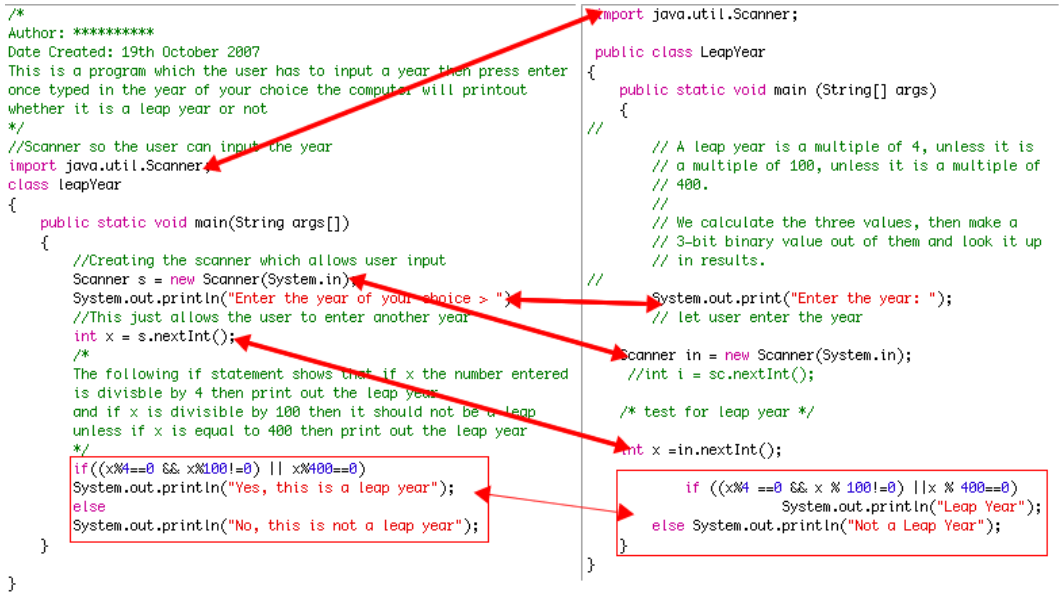


Figure 5.5: Leap Years with A_d measure and $t = 13$

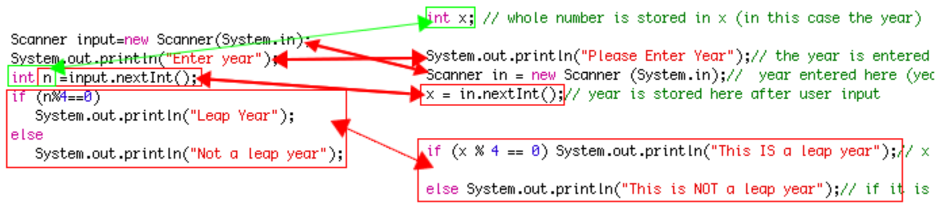


```
/*
Author: **********
Date Created: 19th October 2007
This is a program which the user has to input a year then press enter
once typed in the year of your choice the computer will printout
whether it is a leap year or not
*/
//Scanner so the user can input the year
import java.util.Scanner;
class leapYear
{
    public static void main(String args[])
    {
        //Creating the scanner which allows user input
        Scanner s = new Scanner(System.in);
        System.out.println("Enter the year of your choice > ");
        //This just allows the user to enter another year
        int x = s.nextInt();
        /*
        The following if statement shows that if x the number entered
        is divisble by 4 then print out the leap year
        and if x is divisible by 100 then it should not be a leap
        unless if x is equal to 400 then print out the leap year
        */
        if((x%4==0 && x%100!=0) || x%400==0)
        System.out.println("Yes, this is a leap year");
        else
        System.out.println("No, this is not a leap year");
    }

}
```

```
import java.util.Scanner;

 public class LeapYear
{
    public static void main (String[] args)
    {
//
        // A leap year is a multiple of 4, unless it is
        // a multiple of 100, unless it is a multiple of
        // 400.
        //
        // We calculate the three values, then make a
        // 3-bit binary value out of them and look it up
        // in results.
//
        System.out.print("Enter the year: ");
        // let user enter the year

    Scanner in = new Scanner(System.in);
     //int i = sc.nextInt();

    /* test for leap year */

    int x =in.nextInt();

            if ((x%4 ==0 && x % 100!=0) ||x % 400==0)
                               System.out.println("Leap Year");
        else System.out.println("Not a Leap Year");
    }
}
```

Figure 5.6: Leap Years programs P_8 and P_{14}

[2]Notice also that we have not checked the correctness of the programs



```
Scanner input=new Scanner(System.in);
System.out.println("Enter year");
int n =input.nextInt();
if (n%4==0)
   System.out.println("Leap Year");
else
   System.out.println("Not a leap year");
```

```
int x; // whole number is stored in x (in this case the year)

System.out.println("Please Enter Year");// the year is entered
Scanner in = new Scanner (System.in);//  year entered here (yec
x = in.nextInt();// year is stored here after user input

if (x % 4 == 0) System.out.println("This IS a leap year");// x

else System.out.println("This is NOT a leap year");// if it is
```

Figure 5.7: Leap Years programs P_{17} and P_{27}

4.4 Drawing a Square

In the drawing a square set there are 118 pairs which appear below the threshold. We can see a couple of white ones. Pair (P_{15}, P_{17}) are exact copies with no changes made and pair (P_5, P_{16}), with a similarity of 0.9, are only different because one class has a *public* modifier and one does not.

As an example of another pair from this set see figure 5.9, page 99 which shows that pair (P_1, P_9) contains two very similar programs.

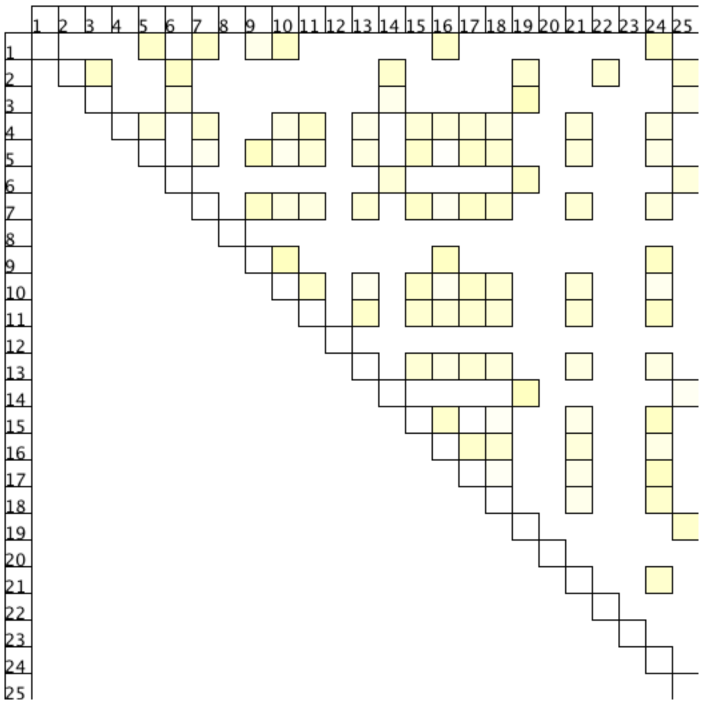
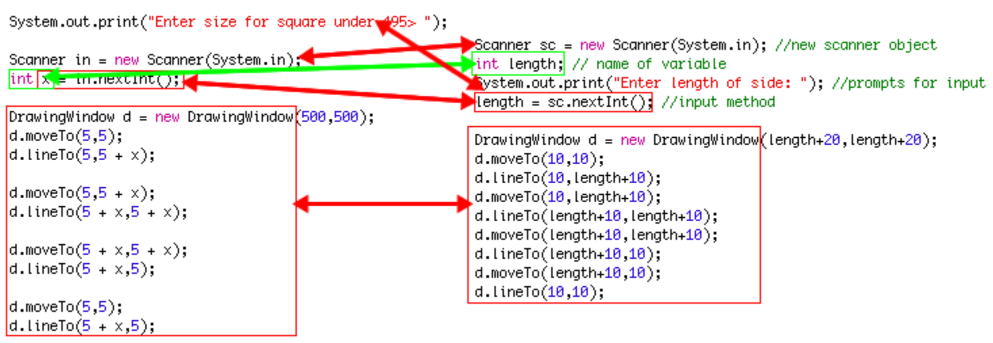


Figure 5.8: Drawing a Square with A_d measure and $t = 13$



```
System.out.print("Enter size for square under 495> ");

Scanner in = new Scanner(System.in);
int x = in.nextInt();

DrawingWindow d = new DrawingWindow(500,500);
d.moveTo(5,5);
d.lineTo(5,5 + x);

d.moveTo(5,5 + x);
d.lineTo(5 + x,5 + x);

d.moveTo(5 + x,5 + x);
d.lineTo(5 + x,5);

d.moveTo(5,5);
d.lineTo(5 + x,5);
```

```
Scanner sc = new Scanner(System.in); //new scanner object
int length; // name of variable
System.out.print("Enter length of side: "); //prompts for input
length = sc.nextInt(); //input method

DrawingWindow d = new DrawingWindow(length+20,length+20);
d.moveTo(10,10);
d.lineTo(10,length+10);
d.moveTo(10,length+10);
d.lineTo(length+10,length+10);
d.moveTo(length+10,length+10);
d.lineTo(length+10,10);
d.moveTo(length+10,10);
d.lineTo(10,10);
```

Figure 5.9: Drawing a Square plagiarised pair (P_1, P_9)

4.5 How Old Are you?

The age set contains 168 pairs which appear below the threshold. Many programs are similar due to their use of an example from the course study guide as the basis for their program. This makes deducing plagiarised pairs from the set difficult.

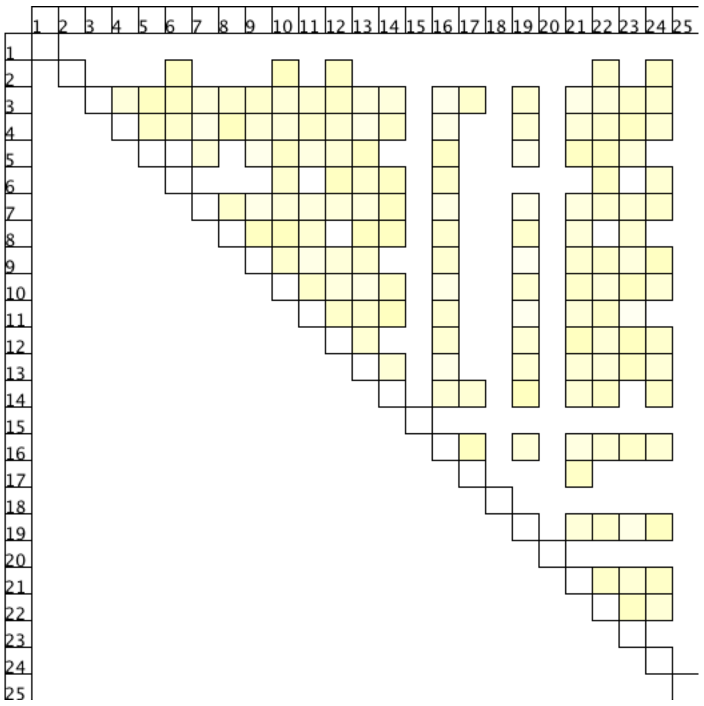
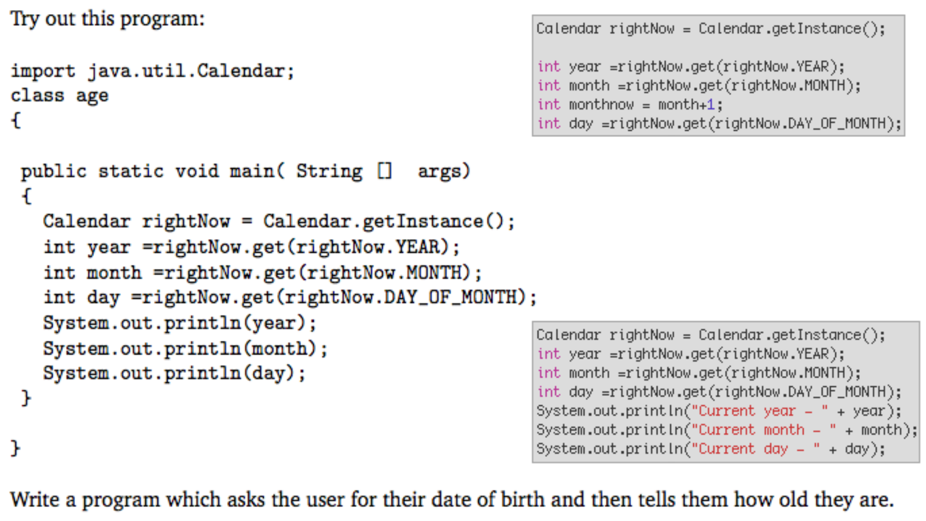


Figure 5.10: How Old Are You? with A_d measure and $t = 13$ (Please note the white squares here are blank i.e. above the threshold)



Try out this program:

```
import java.util.Calendar;
class age
{

 public static void main( String []  args)
 {
   Calendar rightNow = Calendar.getInstance();
   int year =rightNow.get(rightNow.YEAR);
   int month =rightNow.get(rightNow.MONTH);
   int day =rightNow.get(rightNow.DAY_OF_MONTH);
   System.out.println(year);
   System.out.println(month);
   System.out.println(day);
 }

}
```

```
Calendar rightNow = Calendar.getInstance();

int year =rightNow.get(rightNow.YEAR);
int month =rightNow.get(rightNow.MONTH);
int monthnow = month+1;
int day =rightNow.get(rightNow.DAY_OF_MONTH);
```

```
Calendar rightNow = Calendar.getInstance();
int year =rightNow.get(rightNow.YEAR);
int month =rightNow.get(rightNow.MONTH);
int day =rightNow.get(rightNow.DAY_OF_MONTH);
System.out.println("Current year - " + year);
System.out.println("Current month - " + month);
System.out.println("Current day - " + day);
```

Write a program which asks the user for their date of birth and then tells them how old they are.

Figure 5.11: Extract from course study guide [27] with inset (top) extract from P_{19} and (bottom) extract from P_{11}

4.6 Guessing Game

In the guessing game set there are 84 pairs which appear below the threshold. Pair (P_4, P_7) have three characters difference (Figure 5.13, page 101).

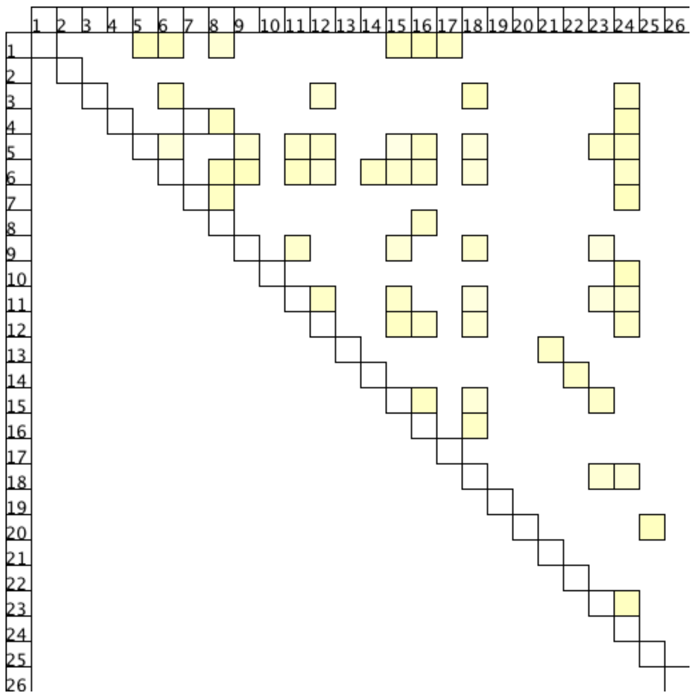
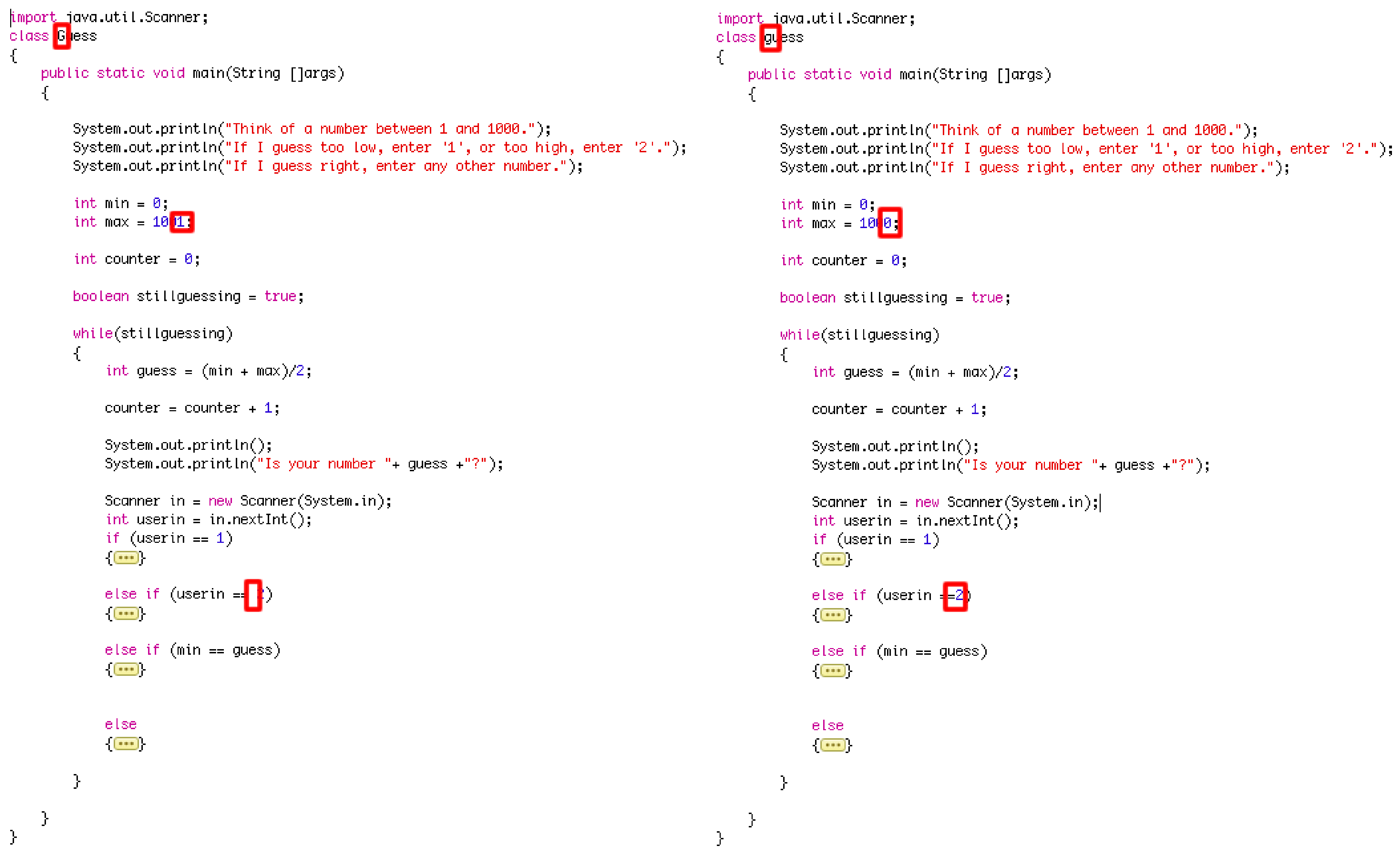


Figure 5.12: Guessing Game with A_d measure and $t = 13$



```
import java.util.Scanner;
class Guess
{
    public static void main(String []args)
    {

        System.out.println("Think of a number between 1 and 1000.");
        System.out.println("If I guess too low, enter '1', or too high, enter '2'.");
        System.out.println("If I guess right, enter any other number.");

        int min = 0;
        int max = 1001;

        int counter = 0;

        boolean stillguessing = true;

        while(stillguessing)
        {
            int guess = (min + max)/2;

            counter = counter + 1;

            System.out.println();
            System.out.println("Is your number "+ guess +"?");

            Scanner in = new Scanner(System.in);
            int userin = in.nextInt();
            if (userin == 1)
            {...}

            else if (userin == 2)
            {...}

            else if (min == guess)
            {...}


            else
            {...}

        }

    }
}
```

```
import java.util.Scanner;
class guess
{
    public static void main(String []args)
    {

        System.out.println("Think of a number between 1 and 1000.");
        System.out.println("If I guess too low, enter '1', or too high, enter '2'.");
        System.out.println("If I guess right, enter any other number.");

        int min = 0;
        int max = 1000;

        int counter = 0;

        boolean stillguessing = true;

        while(stillguessing)
        {
            int guess = (min + max)/2;

            counter = counter + 1;

            System.out.println();
            System.out.println("Is your number "+ guess +"?");

            Scanner in = new Scanner(System.in);
            int userin = in.nextInt();
            if (userin == 1)
            {...}

            else if (userin ==2)
            {...}

            else if (min == guess)
            {...}


            else
            {...}

        }

    }
}
```

Figure 5.13: Plagiarised pair (P_4, P_7) from Guessing Game set with three character differences marked (note: collapsed code is identical)

4.7 Hangman

The hangman set is the smallest with only 8 programs, with 40 pairs appearing below the threshold. There are many false-positives which result in analysis from this set which may be due to the small size.

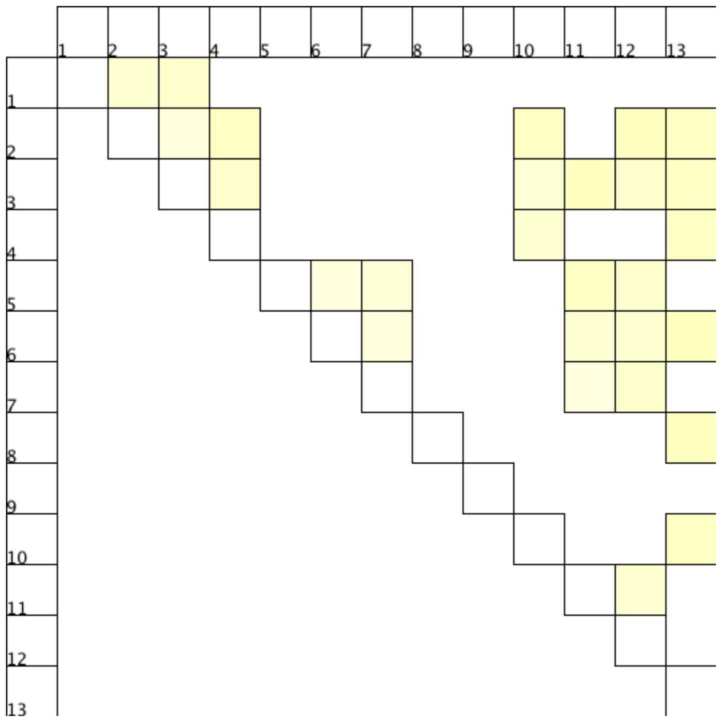


Figure 5.14: Hangman with A_d measure and $t = 13$

4.8 Roman Numerals

In the Roman numerals set there are 37 pairs which appear below the threshold. The pair (P_2, P_6) and (P_{13}, P_{14}) are completely identical - copy and paste plagiarism. Pair (P_{12}, P_{13}) are almost identical with some re-arrangments and renaming, for example their main methods are shown in figure 5.16, page 103. Of course they may just be similar because it is a simple assignment and there aren't many ways of doing it but this demonstrates that the system is capable of detecting similar code.

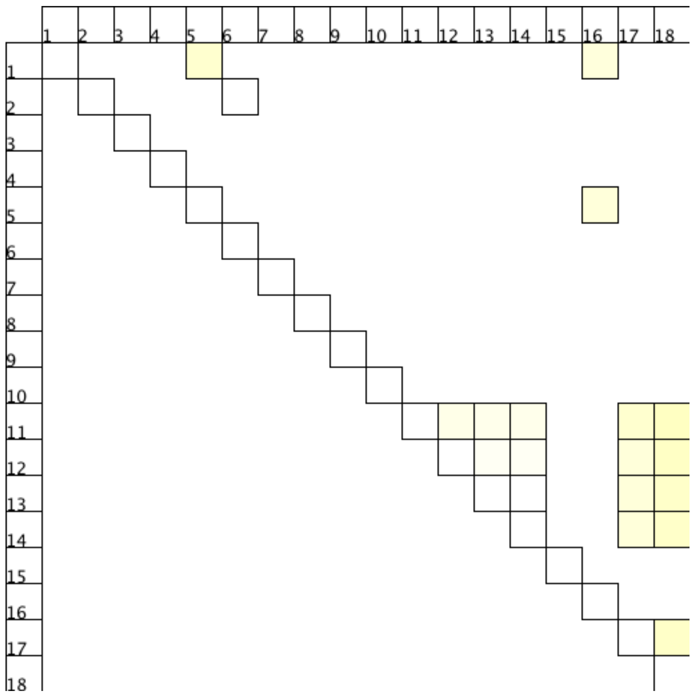
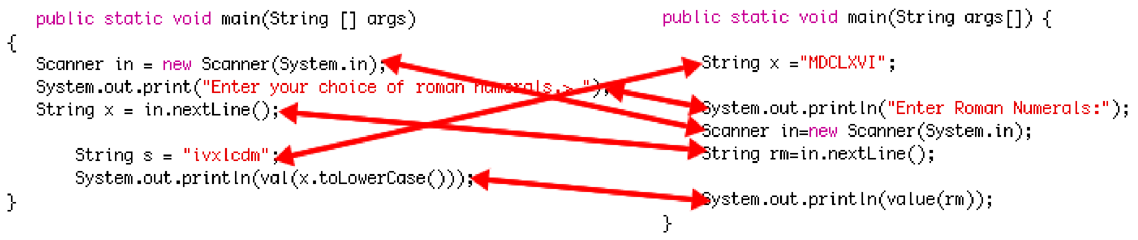


Figure 5.15: Roman Numerals with A_d measure and $t = 13$



```
   public static void main(String [] args)
{
   Scanner in = new Scanner(System.in);
   System.out.print("Enter your choice of roman numerals > ");
   String x = in.nextLine();

       String s = "ivxlcdm";
       System.out.println(val(x.toLowerCase()));
}
```

```
public static void main(String args[]) {
     String x ="MDCLXVI";

     System.out.println("Enter Roman Numerals:");
     Scanner in=new Scanner(System.in);
     String rm=in.nextLine();

     System.out.println(value(rm));
}
```

Figure 5.16: Roman Numerals pair (P_{12}, P_{13}) main methods

5 Conclusion

In this section we have taken some real examples of student submissions to test our system. We started by randomly choosing the maze corpus to pick the best similarity measure and threshold value and we then applied these to the other assignment sets.

Results showed that our system was not as accurate as we would have liked but that the AST node count method was the best similarity measure. We believe structural methods such as AST comparison are better for plagiarism detection than attribute counting methods.

Our results were not as accurate as we hoped but our system did find several plagiarised pairs in the sets which was encouraging. One of the problems with student assignments was highlighted in the Die corpus which is far to simple a program to decide if plagiarism has occurred.

Another problem with student assignment, as shown in the How Old Are You? corpus, is that if students are provided with some parts of the answer as part of the question then all the students programs contain this code. To overcome this problem we must remove certain parts of the code before applying plagiarism detection techniques; this is an idea for further work.

We have demonstrated the usefulness of a graphical representation of the similarities which allowed us to see where plagiarism has occurred in sets which we did not pre-check.

Chapter 6

Conclusion and Future Work

1 Conclusion

We started out evaluating several tools for program transformation and analysis with the aim of finding the 'best' tool. It is hard to define 'best' as many people have different preferences and some tools may be better for some tasks than others. For us the best tool was Eclipse JDT which provided a parser and other Java classes enabling us to statically analyse and transform programs. Other tools could do this too but we found Eclipse JDT the easiest to use. We did not like TXL's functional programming paradigm and preferred a Java based approach. It seemed appropriate that we transformed Java programs using a Java program.

We also liked the Eclipse IDE and will be using this for further development in the future whereas before we started this project we preferred Netbeans. Our development with JavaCC and javac was also performed with Eclipse IDE which allowed us become knowledgeable with its features. The JDT package can also be used separately from Eclipse IDE if it is not your preferred development environment.

Fortunately grammars for Java were available for the parser generator tools and TXL as we could not have seen ourselves undertaking the laborious task of writing a grammar for the Java language when there is an accurate parser implementation available in the form of Eclipse JDT.

When researching tools to evaluate we thought that Sun's java compiler, javac, may be a good tool which we could modify for Java static analysis and transformation tasks as it is a complete parser for the Java language developed by the people who develop Java. Unfortunately we were wrong in this instance and in hindsight realise that it is not a tool for the job. Interestingly we also discovered that in 2006 the Eclipse JDT parser implementation was more accurate than javac [33].

As we were only interested in the Java programming language it was not important for us to be concerned whether or not the tools investigated could be applied to other languages. If, on the other hand, we wanted to transform other languages our choices of tools to research would have been different. One tool that would not have made the list is javac which is only useful in the context of Java parsing, while others would be perfect for parsing any number of languages such as ANTLR or JavaCC. Of course, our favourite tool, Eclipse JDT is only be applicable to the Java language and if we were to have a need for parsing other languages in the future we would choose ANTLR.

Implementing a plagiarism detection system for sets of Java source code allowed us to fully evaluate Eclipse as a static analysis system. Detecting similarities within sets of source code involves statically analysing source code which Eclipse allowed us to do. After implementing the system we realised that Eclipse was a good choice as a basis for the static analysis needs of the application. We also found the Eclipse IDE very good.

Even though our system was not as accurate as we would have liked (MOSS and JPlag have no need to worry just yet!) we learnt a lot about implementing static analysis tools using Eclipse. We now have a basis of knowledge for further work with static analysis problems and our toolset now includes Eclipse to help us solve such problems.

We also discovered the non-trivialness of plagiarism detection in Java source code, and indeed realise how difficult plagiarism detection must be for a non-structured natural language even moreso than a structured language like Java.

Our system was able to discover plagiarism between some pairs of Java programs but results were plagued with false-positives and the range of similarities of known plagiarised pairs was wide. We can suggest many improvements to our plagiarism detection system for further work which we hope can improve its reliability considerably.

Our empirical study also confirmed our hypothesis that structural methods worked better than the simple measure of document length as the AST measure was shown to be the best out of all the measures in the automatic test for the maze corpus.

We confirmed our suspicions that plagiarism detection within a corpus of beginners programming assignments is a difficult task due to the simplicity of the programs. The problem can be clearly seen in the Die assignment corpus whose answer only needs a few lines of code. It is nearly impossible to tell how many, if any, plagiarised pairs there are in such a corpus due to the natural similarity between all submissions.

Another problem we found with plagiarism detection in student assignments, as shown in the How Old Are You? corpus, is that if students are provided with some parts of the answer as part of the question then all the students programs contain this code. This also applies to section of code such as code created by automatic tools such as a GUI builder. We therefore think that the code should be pre-proccessed to remove all the items that we know are going to appear in all the students code such as teachers hints or generated GUI code.

We liked Ellis *et al.* [35] visualisation in their plagiarism detection system for the visualisation of program pairs which, if combined with a more reliable similarity measure than ours, is a good representation of similarities in a set of source code. It easily allows a marker to identify pairs that a similar to each other and we have discovered in the unknown sets such plagiarised pairs by looking at the visualisation though we could not identify all plagiarised pairs due to the wide range of similarities between them using our measures. Indeed we did discover further plagiarised pairs in the other assignment corpuses by using this method.

2 Future Work

2.1 Tools

We believe that a larger survey of static analysis and transformation tools could be undertaken as there are many which we did not cover in our evaluation. As an example two which we did not cover were Netbeans and Stratego.

Netbeans Platform is an alternative to Eclipse and could well be a better alternative, especially as it is developed by Sun who also develop the Java language. The Netbeans Platform[1] is the basis of the Netbeans IDE aimed mainly at Java Development. Netbeans Platform and IDE are written in Java. Program transformations are a requirements of the IDE, to allow for code refactoring [50]. Code refactoring has been a feature of Netbeans IDE for sometime, and its Java Source API has recently merged with the Jackpot Module's API for program transformations [71]. Jackpot also provides a transformation language to allow for simple transformations, and more complicated transformations can be applied via the use of custom Java transformation classes.

Stratego [10] is a program transformation language and set of tools for program transformations designed for transformations of any language. Stratego, like TXL, has its own rewriting language to perform transformations on the input program. The Java Front package [90] supports the implementation of Java transformation systems. The Stratego rewriting language is based around strategic term writing providing rules for creating basic transformation steps, where rewriting strategies are applied which transform terms with respect to a set of rewrite rules [9]. The grammar specification is supplied in Syntax Definition Formalism format and a grammar for Java is part of the Java Front package. Stratego also includes re-usable tools for program transformations such as SDF grammar parsing and pretty printing. Dryad is a library which includes a set of tools for developing transformation systems for Java [32], enabling easier implementation of such systems.

Furthermore we suggest undertaking other tasks to evaluate the systems as it is possible that the results of the tasks we chose were not indicative of results from tasks which we could have chosen. Some tasks were fairly trivial to implement and it may be wise to implement more difficult tasks to fully evaluate the systems.

Though we did not like TXL we were impressed by its power for certain tasks and believe further work should go into studying this system and how it could be applied to our static analysis and transformation needs.

2.2 Plagiarism Detection

Implementation of a plagiarism detection system allowed us to further evaluate our chosen tool Eclipse but proved to be a non-trivial task. Our chosen similarities measures did not detect all plagiarised pairs well and included a fair number of false-positives. We believe that our system could be improved by implementing other similarity measures such as tree-edit-distance or number of common sub-trees to compare program ASTs.

Sasha *et al.* [106] describe such algorithms in detail for exact and approximate matching and Valiente [100] provides some simpler algorithms and implementation examples. Sager *et al.* apply several of Valiente's algorithms to the problem of detecting similar Java classes in the field of clone detection but the same methods can be used for plagiarism detection, as it is, in essence, the same problem. Two methods involved finding the maximum number of common sub-trees, and another involved calculating the tree edit distance.

Finding common sub-trees shows how many parts of two trees are similar whereas the tree edit, which is similar to the string edit distance, is distance between two trees is essentially the cost of the number of operations needed to transform one tree into the other: either substitituion, deletion or insertion.

We believe any further work in this area should look at these structural similarities between programs rather than attribute counting methods.

Though we like our visualisation of program pair similarity further work is needed on the interface to our plagiarism detector for example to provide the ability to see two programs side by side with similarities highlighted.

[1]`http://www.netbeans.org/`

2.3 Code Obfuscation

Code obfuscation is a semantics preserving transformation of source code in such a way that makes it unintelligible to human readers of the code and reverse engineering tools. Plagiarism is a type of obfuscation applied to source code by students but the aim, instead of making the program unreadable, is to create a semantically identical program with enough implementation differences to confound their assignment marker.

Though we only touched on the link between code obfuscation and plagiarism detection in the introduction we believe these fields are closely related and an investigation into code obfuscation and de-obfuscation techniques would be worth while both in its own field and for applications in the field of plagiarism detection. If we can detect an obfuscated program we should be able to also detect a plagiarised program which use simple obfuscation techniques.

As one idea for further work we suggest implementing an automatic plagiariser that takes an original Java source program P a outputs a set of plagiarised versions $\{P_1, P_2, \ldots, P_n\}$ where each P_n is semantically identical to the original but obfuscated in some way so that it may evade detection as plagiarised. Such a system will provide a corpus of source files to test plagiarism detection systems.

2.4 Related Work

Automated Marking Assistance

A system for assisted marking of assignments could be possible by providing a corpus of acceptable source code answers and comparing student submissions against these. Submissions that are less similar to the model answers would be investigated more thoroughly to decide if they are correct. If a submission has a low similarity to a model answer but is in fact completely correct then this could be added to the model answers. Some work has been done in this area [97, 68, 34] which looks for similarities to provide feedback to students instead of plagiarism detection.

In the Die corpus some students submitted more than required with additions such as a graphical representation of a die. Possibly such additions could be identified by plagiarism detection techniques to identify the outstanding students, though a program very different from the few lines of code needed could also be very bad programming.

Clone Detection

Clone detection is basically plagiarism detection for a different purpose. The main purpose of clone detection is to find duplicate code and improve code re-use in large software projects. A lot of work has been done in this area of which Chanchal Kumar Roy and James R. Cordy[2] have recently done an extensive survey of the state of the art [83] and some research has been been done in clone detection by comparing ASTs [7, 52, 84].

[2]James R. Cordy is the creator of TXL

Appendix A

Acronyms

AST Abstract Syntax Tree

ANTLR ANother Tool for Language Recognition

JVM Java Virtual Machine

BNF Backus Naur Form

EBNF Extended Backus Naur Form

JISC Joint Information Systems Committee

GUI Graphical User Interface

JavaCC Java Compiler Compiler

API Application Programmers Interface

Appendix B

Extra Code Listings

1 Test.java

```
/*
 * Test.java
 *
 * Created on Oct 13, 2007, 11:20:20 PM
 *
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package test1;

/**
 *
 * @author james
 */
public class Test {

    static int x = 7;

    public static void main(String...args) {
        int x;
        x = 5 * Test.x;
        x = 5 * 4;

        if(x == 5) {
            hello();
        }else{
            System.out.println("GOODBYE");
        }

        test();
        test3();
        test();
    }


    /* method for calculation ..... */
    public static void test() {

        int z = 5 * Test.x; /*variable x is important */

        System.out.println("test, z=" + z);
        test3();
        test4();

    }

    /*method with a for loop */
    public static void test3() {
        for(int x = 0; x < 5; x++) {
            test4();
        }
    }

    public static void test4() {
        System.out.println("test4");

    }

    public static void hello() {
        System.out.println("HELLO");
    }

    public static void goodbye() { }
}
```

2 Simple Test Program

```
/*
 * Test.java
 *
 * Created on Oct 13, 2007, 11:20:20 PM
 *
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package testpackage;
/**
 *
 * @author james
 */
public class TestA {

    static int x = 7;

    public static void main(String...args) {
        int x;
        x = 5 * TestA.x;
        x = 5 * 4;

        if(x == 5) {
            hello();
        }else{
            System.out.println("GOODBYE");
        }
        test();
        test3();
        test();
    }


    /* method for calculation ..... */
    public static void test() {

        int z = 5 * TestA.x; /*variable x is important */

        System.out.println("test, z=" + z);
        test3();
        test4();

    }

    /*method with a for loop */
    public static void test3() {
        for(int x = 0; x < 5; x++) {
            test4();
        }
    }

    public static void test4() {
        System.out.println("test4");

    }

    public static void hello() {
        System.out.println("HELLO"); t();
    }

public static int t() {
System.out.println("t");
return 1;
}

public boolean a() {
return true;
}
    public static void goodbye() { }
}
```

3 TXL numerical calculator

Listing B.1: TXL numerical expression evaluator grammar

```
% Calculator.Txl - simple numerical expression evaluator

% Part I.  Syntax specification
define program
        [expression]
end define

define expression
        [term]
    |   [expression] [addop] [term]
end define

define term
        [primary]
    |   [term] [mulop] [primary]
end define

define primary
        [number]
    |   ( [expression] )
end define

define addop
        '+
    |   '-
end define

define mulop
        '*
    |   '/
end define
```

Listing B.2: TXL numerical expression evaluator rules

```
% Part 2.  Transformation rules
rule main
    replace [expression]
        E [expression]
    construct NewE [expression]
        E [resolveAddition] [resolveSubtraction] [resolveMultiplication]
          [resolveDivision] [resolveParentheses] [print]
    where not
        NewE [= E]
    by
        NewE
end rule

rule resolveAddition
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule

rule resolveSubtraction
    replace [expression]
        N1 [number] - N2 [number]
    by
        N1 [- N2]
end rule

rule resolveMultiplication
    replace [term]
        N1 [number] * N2 [number]
    by
        N1 [* N2]
end rule

rule resolveDivision
    replace [term]
        N1 [number] / N2 [number]
    by
        N1 [/ N2]
end rule

rule resolveParentheses
    replace [primary]
        ( N [number] )
    by
        N
end rule
```

Appendix C

Results

1 Powerset of Measures Results

```
[A: AST] : 8
[A: AST, W: Loops] : 8
[M: Methods, A: AST] : 8
[M: Methods, A: AST, W: Loops] : 8
[C: Method Invocations, A: AST] : 8
[C: Method Invocations, A: AST, W: Loops] : 8
[C: Method Invocations, M: Methods, A: AST] : 8
[C: Method Invocations, M: Methods, A: AST, W: Loops] : 8
[I: If, A: AST] : 8
[I: If, A: AST, W: Loops] : 8
[I: If, M: Methods, A: AST] : 8
[I: If, M: Methods, A: AST, W: Loops] : 8
[I: If, C: Method Invocations, A: AST] : 8
[I: If, C: Method Invocations, A: AST, W: Loops] : 8
[I: If, C: Method Invocations, M: Methods, A: AST] : 8
[I: If, C: Method Invocations, M: Methods, A: AST, W: Loops] : 8
[V: Variables, A: AST] : 8
[V: Variables, A: AST, W: Loops] : 8
[V: Variables, M: Methods, A: AST] : 8
[V: Variables, M: Methods, A: AST, W: Loops] : 8
[V: Variables, C: Method Invocations, A: AST] : 8
[V: Variables, C: Method Invocations, A: AST, W: Loops] : 8
[V: Variables, C: Method Invocations, M: Methods, A: AST] : 8
[V: Variables, C: Method Invocations, M: Methods, A: AST, W: Loops] : 8
[V: Variables, I: If, A: AST] : 8
[V: Variables, I: If, A: AST, W: Loops] : 8
[V: Variables, I: If, M: Methods, A: AST] : 8
[V: Variables, I: If, M: Methods, A: AST, W: Loops] : 8
[V: Variables, I: If, C: Method Invocations, A: AST] : 8
[V: Variables, I: If, C: Method Invocations, A: AST, W: Loops] : 8
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST] : 8
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, W: Loops] : 8
[A: AST, F: Fingerprint] : 5
[A: AST, W: Loops, F: Fingerprint] : 5
[M: Methods, A: AST, F: Fingerprint] : 5
[M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[C: Method Invocations, A: AST, F: Fingerprint] : 5
[C: Method Invocations, A: AST, W: Loops, F: Fingerprint] : 5
[C: Method Invocations, M: Methods, A: AST, F: Fingerprint] : 5
[C: Method Invocations, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[I: If, A: AST, F: Fingerprint] : 5
[I: If, A: AST, W: Loops, F: Fingerprint] : 5
```

```
[I: If, M: Methods, A: AST, F: Fingerprint] : 5
[I: If, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[I: If, C: Method Invocations, A: AST, F: Fingerprint] : 5
[I: If, C: Method Invocations, A: AST, W: Loops, F: Fingerprint] : 5
[I: If, C: Method Invocations, M: Methods, A: AST, F: Fingerprint] : 5
[I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, A: AST, F: Fingerprint] : 5
[V: Variables, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, M: Methods, A: AST, F: Fingerprint] : 5
[V: Variables, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, C: Method Invocations, A: AST, F: Fingerprint] : 5
[V: Variables, C: Method Invocations, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, C: Method Invocations, M: Methods, A: AST, F: Fingerprint] : 5
[V: Variables, C: Method Invocations, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, I: If, A: AST, F: Fingerprint] : 5
[V: Variables, I: If, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, I: If, M: Methods, A: AST, F: Fingerprint] : 5
[V: Variables, I: If, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, I: If, C: Method Invocations, A: AST, F: Fingerprint] : 5
[V: Variables, I: If, C: Method Invocations, A: AST, W: Loops, F: Fingerprint] : 5
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, F: Fingerprint] : 5
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, F: Fingerprint] : 5
[C: Method Invocations, F: Fingerprint] : 4
[C: Method Invocations, W: Loops, F: Fingerprint] : 4
[C: Method Invocations, M: Methods, F: Fingerprint] : 4
[C: Method Invocations, M: Methods, W: Loops, F: Fingerprint] : 4
[I: If, C: Method Invocations, F: Fingerprint] : 4
[I: If, C: Method Invocations, W: Loops, F: Fingerprint] : 4
[I: If, C: Method Invocations, M: Methods, F: Fingerprint] : 4
[I: If, C: Method Invocations, M: Methods, W: Loops, F: Fingerprint] : 4
[V: Variables, F: Fingerprint] : 4
[V: Variables, W: Loops, F: Fingerprint] : 4
[V: Variables, M: Methods, F: Fingerprint] : 4
[V: Variables, M: Methods, W: Loops, F: Fingerprint] : 4
[V: Variables, C: Method Invocations, F: Fingerprint] : 4
[V: Variables, C: Method Invocations, W: Loops, F: Fingerprint] : 4
[V: Variables, C: Method Invocations, M: Methods, F: Fingerprint] : 4
[V: Variables, C: Method Invocations, M: Methods, W: Loops, F: Fingerprint] : 4
[V: Variables, I: If, F: Fingerprint] : 4
[V: Variables, I: If, W: Loops, F: Fingerprint] : 4
[V: Variables, I: If, M: Methods, F: Fingerprint] : 4
[V: Variables, I: If, M: Methods, W: Loops, F: Fingerprint] : 4
[V: Variables, I: If, C: Method Invocations, F: Fingerprint] : 4
[V: Variables, I: If, C: Method Invocations, W: Loops, F: Fingerprint] : 4
[V: Variables, I: If, C: Method Invocations, M: Methods, F: Fingerprint] : 4
[V: Variables, I: If, C: Method Invocations, M: Methods, W: Loops, F: Fingerprint] : 4
[F: Fingerprint] : 3
[W: Loops, F: Fingerprint] : 3
[M: Methods, F: Fingerprint] : 3
[M: Methods, W: Loops, F: Fingerprint] : 3
[I: If, F: Fingerprint] : 3
[I: If, W: Loops, F: Fingerprint] : 3
[I: If, M: Methods, F: Fingerprint] : 3
[I: If, M: Methods, W: Loops, F: Fingerprint] : 3
[V: Variables, I: If, C: Method Invocations, W: Loops] : 3
[V: Variables, I: If, C: Method Invocations, M: Methods, W: Loops] : 3
[L: Document Length, F: Fingerprint] : 2
[W: Loops, L: Document Length, F: Fingerprint] : 2
[A: AST, L: Document Length, F: Fingerprint] : 2
```

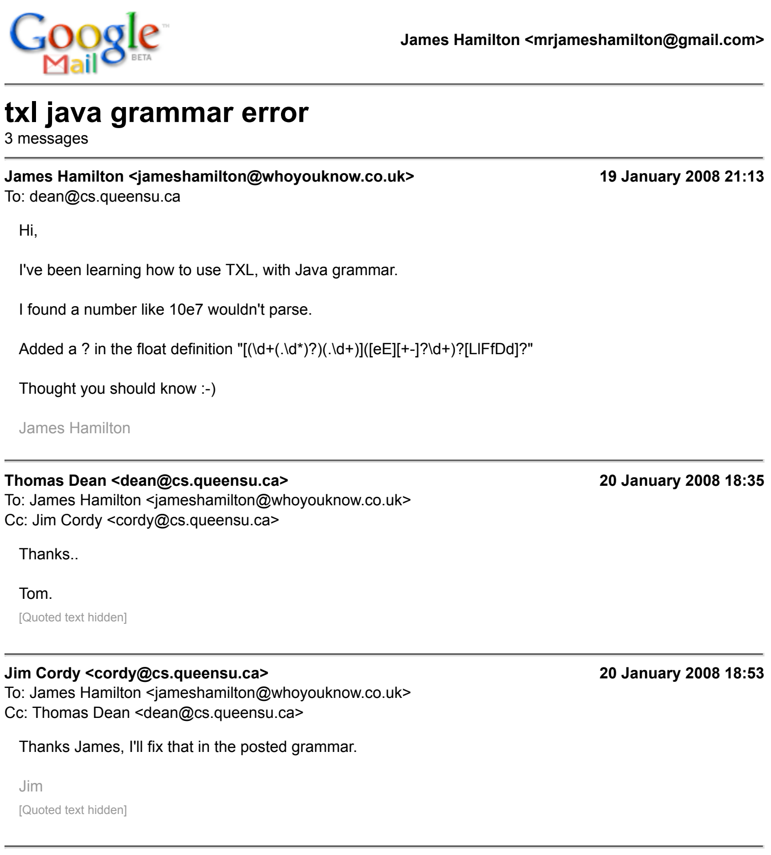
```
[A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[M: Methods, L: Document Length, F: Fingerprint] : 2
[M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, W: Loops, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, A: AST, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, M: Methods, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, L: Document Length, F: Fingerprint] : 2
[I: If, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, A: AST, L: Document Length, F: Fingerprint] : 2
[I: If, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, M: Methods, L: Document Length, F: Fingerprint] : 2
[I: If, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[I: If, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, W: Loops] : 2
[I: If, C: Method Invocations, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, A: AST, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, M: Methods, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, M: Methods, W: Loops] : 2
[I: If, C: Method Invocations, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, L: Document Length, F: Fingerprint] : 2
[V: Variables, W: Loops] : 2
[V: Variables, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, M: Methods, L: Document Length, F: Fingerprint] : 2
[V: Variables, M: Methods, W: Loops] : 2
[V: Variables, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, W: Loops] : 2
[V: Variables, C: Method Invocations, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, M: Methods, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, M: Methods, W: Loops] : 2
[V: Variables, C: Method Invocations, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprin
[V: Variables, I: If] : 2
[V: Variables, I: If, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, W: Loops] : 2
[V: Variables, I: If, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, M: Methods] : 2
[V: Variables, I: If, M: Methods, L: Document Length, F: Fingerprint] : 2
```

```
[V: Variables, I: If, M: Methods, W: Loops] : 2
[V: Variables, I: If, M: Methods, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, M: Methods, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, M: Methods, A: AST, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, C: Method Invocations] : 2
[V: Variables, I: If, C: Method Invocations, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, C: Method Invocations, W: Loops, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, C: Method Invocations, A: AST, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, C: Method Invocations, A: AST, W: Loops, L: Document Length, F: Fingerprint] :
[V: Variables, I: If, C: Method Invocations, M: Methods] : 2
[V: Variables, I: If, C: Method Invocations, M: Methods, L: Document Length, F: Fingerprint] : 2
[V: Variables, I: If, C: Method Invocations, M: Methods, W: Loops, L: Document Length, F: Fingerprint
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, L: Document Length, F: Fingerprint]
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length, F: Fin
[W: Loops] : 1
[A: AST, L: Document Length] : 1
[A: AST, W: Loops, L: Document Length] : 1
[M: Methods, W: Loops] : 1
[M: Methods, A: AST, L: Document Length] : 1
[M: Methods, A: AST, W: Loops, L: Document Length] : 1
[C: Method Invocations, A: AST, L: Document Length] : 1
[C: Method Invocations, A: AST, W: Loops, L: Document Length] : 1
[C: Method Invocations, M: Methods, A: AST, L: Document Length] : 1
[C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[I: If] : 1
[I: If, W: Loops] : 1
[I: If, A: AST, L: Document Length] : 1
[I: If, A: AST, W: Loops, L: Document Length] : 1
[I: If, M: Methods] : 1
[I: If, M: Methods, W: Loops] : 1
[I: If, M: Methods, A: AST, L: Document Length] : 1
[I: If, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[I: If, C: Method Invocations, A: AST, L: Document Length] : 1
[I: If, C: Method Invocations, A: AST, W: Loops, L: Document Length] : 1
[I: If, C: Method Invocations, M: Methods, A: AST, L: Document Length] : 1
[I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, A: AST, L: Document Length] : 1
[V: Variables, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, M: Methods, A: AST, L: Document Length] : 1
[V: Variables, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, C: Method Invocations, A: AST, L: Document Length] : 1
[V: Variables, C: Method Invocations, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, C: Method Invocations, M: Methods, A: AST, L: Document Length] : 1
[V: Variables, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, I: If, A: AST, L: Document Length] : 1
[V: Variables, I: If, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, I: If, M: Methods, A: AST, L: Document Length] : 1
[V: Variables, I: If, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, I: If, C: Method Invocations, A: AST, L: Document Length] : 1
[V: Variables, I: If, C: Method Invocations, A: AST, W: Loops, L: Document Length] : 1
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, L: Document Length] : 1
[V: Variables, I: If, C: Method Invocations, M: Methods, A: AST, W: Loops, L: Document Length] : 1
[] : 0
[L: Document Length] : 0
[W: Loops, L: Document Length] : 0
[M: Methods] : 0
[M: Methods, L: Document Length] : 0
[M: Methods, W: Loops, L: Document Length] : 0
[C: Method Invocations] : 0
```

```
[C: Method Invocations, L: Document Length] : 0
[C: Method Invocations, W: Loops] : 0
[C: Method Invocations, W: Loops, L: Document Length] : 0
[C: Method Invocations, M: Methods] : 0
[C: Method Invocations, M: Methods, L: Document Length] : 0
[C: Method Invocations, M: Methods, W: Loops] : 0
[C: Method Invocations, M: Methods, W: Loops, L: Document Length] : 0
[I: If, L: Document Length] : 0
[I: If, W: Loops, L: Document Length] : 0
[I: If, M: Methods, L: Document Length] : 0
[I: If, M: Methods, W: Loops, L: Document Length] : 0
[I: If, C: Method Invocations] : 0
[I: If, C: Method Invocations, L: Document Length] : 0
[I: If, C: Method Invocations, W: Loops, L: Document Length] : 0
[I: If, C: Method Invocations, M: Methods] : 0
[I: If, C: Method Invocations, M: Methods, L: Document Length] : 0
[I: If, C: Method Invocations, M: Methods, W: Loops, L: Document Length] : 0
[V: Variables] : 0
[V: Variables, L: Document Length] : 0
[V: Variables, W: Loops, L: Document Length] : 0
[V: Variables, M: Methods] : 0
[V: Variables, M: Methods, L: Document Length] : 0
[V: Variables, M: Methods, W: Loops, L: Document Length] : 0
[V: Variables, C: Method Invocations] : 0
[V: Variables, C: Method Invocations, L: Document Length] : 0
[V: Variables, C: Method Invocations, W: Loops, L: Document Length] : 0
[V: Variables, C: Method Invocations, M: Methods] : 0
[V: Variables, C: Method Invocations, M: Methods, L: Document Length] : 0
[V: Variables, C: Method Invocations, M: Methods, W: Loops, L: Document Length] : 0
[V: Variables, I: If, L: Document Length] : 0
[V: Variables, I: If, W: Loops, L: Document Length] : 0
[V: Variables, I: If, M: Methods, L: Document Length] : 0
[V: Variables, I: If, M: Methods, W: Loops, L: Document Length] : 0
[V: Variables, I: If, C: Method Invocations, L: Document Length] : 0
[V: Variables, I: If, C: Method Invocations, W: Loops, L: Document Length] : 0
[V: Variables, I: If, C: Method Invocations, M: Methods, L: Document Length] : 0
[V: Variables, I: If, C: Method Invocations, M: Methods, W: Loops, L: Document Length] : 0
```

Appendix D

Correspondence



Google Mail BETA

James Hamilton <mrjameshamilton@gmail.com>

txl java grammar error

3 messages

James Hamilton <jameshamilton@whoyouknow.co.uk> **19 January 2008 21:13**
To: dean@cs.queensu.ca

Hi,

I've been learning how to use TXL, with Java grammar.

I found a number like 10e7 wouldn't parse.

Added a ? in the float definition "[(\d+(.\d*)?)(.\d+)]([eE][+-]?\d+)?[LlFfDd]?"

Thought you should know :-)

James Hamilton

Thomas Dean <dean@cs.queensu.ca> **20 January 2008 18:35**
To: James Hamilton <jameshamilton@whoyouknow.co.uk>
Cc: Jim Cordy <cordy@cs.queensu.ca>

Thanks..

Tom.

[Quoted text hidden]

Jim Cordy <cordy@cs.queensu.ca> **20 January 2008 18:53**
To: James Hamilton <jameshamilton@whoyouknow.co.uk>
Cc: Thomas Dean <dean@cs.queensu.ca>

Thanks James, I'll fix that in the posted grammar.

Jim

[Quoted text hidden]

Figure D.1: Email advising of TXL Java Grammar Error

Bibliography

[1] Computer plagiarism 'threatens the value of degrees'. 2001. `http://www.telegraph.co.uk/news/uknews/1346324/article.html`.

[2] Many degree students 'cheating'. 2007. `http://news.bbc.co.uk/1/hi/education/6363647.stm`.

[3] University cheats 'not expelled'. 2008. `http://news.bbc.co.uk/1/hi/education/7434277.stm`.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[5] C. T. Bailey and W. L. Dingee. A software study using halstead metrics. *SIGMETRICS Perform. Eval. Rev.*, 10(1):189–197, 1981.

[6] Tom Ball. Hacking javac. `http://weblogs.java.net/blog/tball/archive/2006/09/hacking_javac.html`, 2006.

[7] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Russel Impagliazzo Steven Rudich Amit Sahai Salil Vadhan Ke Yang Boaz Barak, Oded Goldreich. On the (im)possiblity of obfuscating programs.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.16: components for transformation systems. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 95–99, New York, NY, USA, 2006. ACM.

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008.

[11] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph, 1998.

[12] Jon Byous. Java technology: The early years. 2003.

[13] Ruth Barret Caroline Lyon and James Malcolm. Plagiarism is easy, but also easy to detect. *Plagiary: Cross-Disciplinary Studies in Plagiarism, Fabrication, and Falsification*, 2006.

[14] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection.

[15] Xin Chen, Ming Li, Brian Mckinnon, and Amit Seker. A theory of uncheatable program plagiarism detection and its practical implementation.

[16] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[17] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Finding similarities in source code through factorization. In *8th Workshop on Language Descriptions, Tools and Applications (LDTA'08)*, Electronic Notes in Theoretical Computer Science, 2008. (15 pp.).

[18] Nom Chomsky. Three models for the description of language. 1956.

[19] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns, 2003.

[20] Paul Clough. Plagiarism in natural and programming languages: an overview of current tools and technologies, 2000.

[21] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks, 2003.

[22] Christian Collberg, Ginger Myles, and Michael Stepp. Cheating cheating detectors.

[23] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. July 1997.

[24] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.

[25] J. Cordy. Txl – a language for programming language tools and applications. 2004.

[26] Fintan Culwin and Thomas Lancaster. A review of electronic services for plagiarism detection in student submissions. 2000.

[27] Sebastian Danicic. *Introduction to Java and Object Oriented Programming.* 2007.

[28] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *Java(TM) Developer's Guide to Eclipse, The (2nd Edition).* Addison-Wesley Professional, 2004.

[29] Robert B. K. Dewar and Edmond Schonberg. Computer science education: Where are the software engineers of tomorrow? *CrossTalk, The Journal of Defense Software Engineering*, 21(1):28–30, January 2008.

[30] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. A plagiarism detection system. In *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 21–25, New York, NY, USA, 1981. ACM.

[31] Timothy B. D'Orazio. *Programming in C++: Lessons and Applications.*

[32] Dryad. Dryad - program transformation system for java - stratego library. `http://www.program-transformation.org/Stratego/TheDryad`.

[33] Bruce Eckel and Kirk Pepperdine. Jdt more correct than javac. `http://www.theserverside.com/news/thread.tss?thread_id=38644`, 2006.

[34] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3):1, 2003.

[35] Matt G. Ellis and Claude W. Anderson. Plagiarism detection in computer code. 2005.

[36] M. Ernst. Static and dynamic analysis: synergy and duality, 2003.

[37] Jim Idle et al. Antlr target languages, 2008. `http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets`.

[38] Terrance Parr et al. Antlr grammar list, 2008. `http://www.antlr.org/grammar/list`.

[39] J. A. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. 1987.

[40] David Falkenstein. Im not in denial, im in a hurry. 2006.

[41] Anna MacLeod & Thomas Lancaster Fintan Culwin. Source code plagiarism in uk he, computing schools, issues, attitudes and tools, 2001.

[42] Manuel Freire, Manuel Cebrian, and Emilio del Rosal. Ac: An integrated source code plagiarism detection environment, 2007.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[44] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[45] M. H. Halstead. Natural laws controlling algorithm structure? *SIGPLAN Not.*, 7(2):19–26, 1972.

[46] Maurice H. Halstead. *Elements of software science (Operating and programming systems series)*. Elsevier, 1977.

[47] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[48] R.W.; Irving. Plagiarism and collusion detection using the smith-waterman algorithm, 2004.

[49] John T. Waldron James F. Power. Recent advances in java technology theory, application, implementation.

[50] Petr Zajac Jan Be?i?ka and Petr H?ebejk. Using java 6 compiler as a refactoring and an analysis engine. 2007.

[51] Thomas Vandrunen Jens. Visitor-oriented programming.

[52] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Eleftherios Koutsoos Stephen C. North John Ellson, Emden R. Gansner and Gordon Woodhull. Graphviz and dynagraph static and dynamic graph drawing tools.

[54] Edward L. Jones. Metrics based plagarism monitoring. *J. Comput. Small Coll.*, 16(4):253–261, 2001.

[55] Nahomi Kikuchi and Tohru Kikuno. Improving the testing process by program static analysis. *apsec*, 0:195, 2001.

[56] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[57] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[58] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[59] Ronald J. Leach. Using metrics to evaluate student programs. *SIGCSE Bull.*, 27(2):41–43, 1995.

[60] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[61] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly, 2003.

[62] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, New York, NY, USA, 2006. ACM.

[63] Sara Capecchi Lorenzo Bettini and Betti Venneri. Translating double dispatch into single dispatch. *Electronic Notes in Theoretical Computer Science*, 138, 2005.

[64] David B. Loveman. Program improvement by source-to-source transformation. *J. ACM*, 24(1):121–145, 1977.

[65] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages.

[66] Michael Madden and Desmond Chambers. Evaluation of student attitudes to learning the java language.

[67] China Martens. The bulk of java is open sourced. 2007.

[68] Susan A. Mengel and Vinay Yerramilli. A case study of the static analysis of the quality of novice student programs. *SIGCSE Bull.*, 31(1):78–82, 1999.

[69] Gilad Mishne and Maarten de Rijke. Source code retrieval using conceptual similarity.

[70] Indranil Nandy. Halsteads operators and operands.

[71] Netbeans. Netbeans jackpot module.

[72] E. Nurvitadhi, E. Nurvitadhi, and C. Cook. Do class comments aid java program understanding? *fie*, 1:T3C13–17, 2003.

[73] K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, 8(4):30–41, 1976.

[74] Tim Ottinger. Meaningful names. http://tottinge.blogsome.com/meaningfulnames/, 1997.

[75] Patrick Page, Rudolf K. Keller, and Reinhard Schauer. A javacc parser for the uml-based cdif transfer format.

[76] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, May 2007.

[77] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[78] Terrance Parr. Javacc vs antlr. `http://compilers.iecc.com/comparch/article/97-09-10`.

[79] Mark O. Pendergast. Teaching introductory programming to is students: Java problems and pitfalls, 2006.

[80] L. Prechelt, G. Malpohl, and Michael Philippsen. Finding plagiarism among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[81] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, nov 2002. `http://www.jucs.org/jucs_8_11/finding_plagiarisms_among_a`.

[82] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[83] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research, 2007.

[84] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 65–71, New York, NY, USA, 2006. ACM.

[85] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *Information and Media Technologies*, 1(1):134–146, 2006.

[86] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. 2003.

[87] Michael I. Schwartzbach. Lecture notes on static analysis. 2008.

[88] SCO. Caldera's complaint - caldera v. ibm. `http://www.groklaw.net/article.php?story=20040704170212250`.

[89] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[90] Stratego.

[91] Michael Studman. Comparing asts of the two java 1.5 grammars. `http://osdir.com/ml/parsers.antlr-interest/2004-10/msg00219.html`, 2004.

[92] Sun. Java 1.5 language features. `"http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#lang"`.

[93] Sun. Jsr199 - java compiler api.

[94] Sun. Java 1.6 language features. `http://www.mydigitallife.info/2006/08/25/java-se-6-java-platform-standard-edition-6-features-and-enhancements/`, 2006.

[95] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, February 2004. Innsbruck, Austria.

[96] Olivier Thomann IBM Ottawa Lab Thomas Kuhn, Eye Media GmbH. Abstract syntax tree. 2006.

[97] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In *ACE '04: Proceedings of the sixth conference on Australasian computing education*, pages 317–325, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[98] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[99] Paul Tyma. Why are we using java again? *Commun. ACM*, 41(6):38–42, 1998.

[100] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[101] Eelco Visser et al. The Program Transformation Wiki. `http://www.program-transformation.org`.

[102] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, 2006.

[103] G. Whale. Identification of program similarity in large populations. 1990.

[104] Michael J. Wise. String similarity via greedy string tiling and running karp?rabin matching. 1993.

[105] Bob Zeidman. What, exactly, is software plagiarism? *Intellectual Property Today*, 2007.

[106] Kaizhong Zhang, Dennis Shasha, and Jason Tsong-Li Wang. Approximate tree matching in the presence of variable length don't cares. *J. Algorithms*, 16(1):33–66, 1994.