Comparing Java programs: syntactic and contextual semantic differences

Siviling. thesis

Hung Huynh

University of Oslo
January 2005

Abstract

During development programmers are often faced with the need to compare two programs, or more exactly two different versions of a program. Line-based tools like *diff* often produce imprecise or too verbose outcome when applied to programs because they do not recognize that programs are bound to a rigid structure. Other tools have a different approach, such as comparing declarations. But they will only report differences in the API. We give in this thesis a tool that combines the technique of these two approaches. We take both declarations and syntax into account, as well as contextual semantic differences. Our tool recognizes for example the context of methods; some methods are declared in the class they are implemented, others are inherited from interfaces and some override the classes' parent's methods.

This thesis rests heavily on the work of Yang and we have created variants of Yang's algorithms and extended them so they fit our purpose. We introduce algorithms that handle both ordered and unordered nodes in an abstract syntax tree. Ordered nodes in an abstract syntax tree denote for example statements in a method body. Each node represents one statement and the order of the statement is significant for the computation of that method. Therefore our algorithms sometimes must respect the order of the nodes when comparing them. But in the other hand, we do not need to respect the order of nodes when they represent for example methods. The order of the methods in class is not important in Java. Our tool is adapted especially to object-oriented aspects of Java, such as inheritance.

We implement the algorithms in the functional language Haskell and use the Strafunski libraries for generic programming to transform Java programs into abstract syntax trees and partly to traverse the trees. And since we exploit the fact that programs are bound to a rigid syntactic structure and that the comparison takes knowledge of Java into account, we are able to create improved and useful difference reports regarding syntax, object-orientated constructs and their semantics.

Table of contents

1. Introduction

In this chapter we explain the starting point for this thesis. The tool described in this thesis is implemented in the functional programming language Haskell. We give therefore a small introduction of Haskell. Last are the descriptions of the remaining chapters.

1.1. Motivation

Programmers often work in groups and several persons might work on the same program. The need for a tool for comparing versions of a program is therefore big. They need a tool that can identify important and useful changes and not be confused by for example the order of methods, white-space and inheritance.

The Unix tool *diff*[1] does not produce desirable outputs when comparing programs since its purpose is to compare text. Ordinary text has a different and simpler structure than programs. The tool *diff* compares text line-by-line, where the order is significant. Programs are not suited to be broken down into lines, but rather into methods, constructor, fields etc.

The tool *JDiff*[2] in the other hand has a different approach. It only compares the declarations of methods and ignores the implementations of them. The main difference between *JDiff* and *diff* is that *JDiff* has knowledge of programming language Java. Therefore it recognizes fields and methods. The tool *JDiff* is most suited for comparing distributions of API of large software systems. Since the implementations are not a part of the comparison, *JDiff* is not suited for the daily usage of programmers.

Our approach is to combine *JDiff's* declaration comparison with *diff's* line-based comparison and comparing abstract syntax trees instead of lines. Our tool works in the higher level of abstraction, such as inheritance between interfaces and classes, and also in the details of the implementation, such as variables and statements. It also exploits the fact that Java is bound to a rigid syntactic structure. The comparison algorithms also have knowledge of Java syntax as well as some understanding of semantic. The combination of the two approaches and the knowledge of Java make a powerful and useful tool for programmers, and hopefully improve their efficiency on software maintenance.

The difference report of two programs is the same as the least editing script from an older version to a newer. The problem of finding the least editing script is equivalent with finding the longest common subsequence of two sequences. Our approach of comparing Java programs is to compare abstract syntax tree representations of the two versions. Therefore our job is not just finding the longest common subsequence, but also finding the equivalent for trees, the largest common subtree.

There are in general several algorithms for finding the least editing script between two sequences and for finding the largest common subtree. But the shortest editing scripts and the

[1] http://www.gnu.org/software/diffutils/diffutils.html
[2] http://www.jdiff.org/

largest common subtrees may not be the best, or the ones that we are after. We adjust these algorithms to suit our approaches and the programming language Java.

1.2. Haskell and functional programming

The implementation of our tool is done in Haskell, which is a general purpose, non-strict and purely functional programming language. Functional programming is based on the model of finding the value of an expression and it therefore uses equations.

The idea of Haskell was born in 1987 at the conference on Functional Programming Language and Computer Architecture (FPCA '87) in the USA. At that point there existed a dozen or so functional languages, all very similar. At a meeting it was decided that a new functional language should be designed to unify the existing functional languages and to encourage the use of functional languages. The result was Haskell, named after the logician Haskell Brooks Curry.

After four iterations, the 1997 Haskell Workshop in the Netherlands felt that a stable variant of Haskell was needed. The result was Haskell 98, which we use in this thesis and which we refer to when mentioning Haskell. Older versions of Haskell are now obsolete.

We give in section 1.2.1 an introduction to Haskell. Our aim is not to teach Haskell or functional programming, but to give a foundation for understanding the implementation we describe in later chapters.

1.2.1. A short introduction to Haskell

First of all, all computation in Haskell is done via the evaluation of expressions to yield values. All values are associated with a type. Intuitively we think of types as sets of values. For example, the values 1, 10 and 100 are associated with the type integer, and '*a*', '*b*' and '*c*' are associated with characters. Functions are first-class citizens, which means that may be passed on as arguments, returned as results, elements in data structures etc. Types on the other hand are not.

We are able to define our own types in Haskell using a data declaration. The type *Parent* are defined as follow:

data Parent = Mother | Father

The type *Parent* has exactly two values; *Mother* or *Father*. Types can also be defined recursive, as in trees:

data Tree = Empty | Node Int [Tree]

We have defined a tree that is either *Empty* or consist of a *Node*. Note that "*Node*" and "*Empty*" are constructors. The type *Node* consists of an integer and a list of *Trees*. We use this construction as basis for our algorithms and functions in chapter 4 and 5.

Functions can be declared in several ways. The simplest way of declaring a function is just by equation:

$$add\ x\ y = x+y$$

Another kind of declaration of the *add* function above is type signature declaration:

$$add :: Integer \text{ -> } Integer \text{-> } Integer$$

The "::" can be read as "has type". The function *add* is an example of a curried function. Applying the first argument to function *add* yields another function which is applied to the second argument. The type of the function *add*, Integer -> Integer -> Integer is equivalent to Integer -> (Integer -> Integer), where the "->" is right associated. Here is an uncurried version of the function *add*.

$$add\ (x,y) = x + y$$

Arrays in Haskell are formed from a pair of bounds (the beginning index and the end index) and a list of index-value pairs. Here is an array of length 10 containing the squares of numbers from one to ten.

$$squares = array\ (1, 10)\ [\ (\ i, i*i\ )\ |\ i <\text{- } [1..10]]$$

Here *"(1, 10)"* denotes the pair of bounds, where 1 is the starting point of the array and 10 is the last index. The pair *"(i, i*i)"* is the index and value, respectively. The last part of the array creation "i <- [1..10]" tells us that the range of index i is from 1 to 10. Basically the line above is says that the array *squares* is of length 10 and index i = i*i for i $\in$ [1..10]. Subscripting is performed with the infix operator "!". For example, *squares*!2 = 4. Note that subscripting lists is performed with the infix operator "!!" and the first element of lists has always index 0.

Constructing matrices is virtually the same as arrays. We only need to replace any singular indexes with a pair of indexes. The pair of bounds of a matrix has therefore the form ((k, l), (m, n)). The size of the matrix is then (m-k) × (n-l).

Many problems in Haskell are solved using recursive functions. These functions are defined using themselves in the definition.

$$factorial\ 0 = 1$$
$$factorial\ 1 = 1$$
$$factorial\ n = n + factorial\ (n-1)$$

This function will successfully compute factorial for all positive integers. The two first lines are a specific case of the function. It dictates that if the function gets the value 0 or 1 as argument, then it returns the value 1. The successor lines are then not computed. Another way

of handle specific cases in Haskell is using guards. Here is the same factorial function as above, but with guards:

```
factorial n
        | n = = 0 = 1
        | n = = 1 = 1
        | otherwise = n + factorial (n-1)
```

The first two guards checks whether the argument is 0 or 1, in which case it returns the value 1. The value of *otherwise* is always true and the last guard is therefore the default guard. The infix operator "= =" is the equality operator, it takes two arguments of the same type and returns either false or true, which denotes the equality of the two arguments.

One of the most powerful features of Haskell is pattern matching. Haskell automatically recognizes the internal representation of an abstract data type. We use pattern matching extensively in our functions. For example, let N be a node as describe earlier and have the application value "*Node 2 []*", then:

f (Node intN subTreesN) and f N

By using pattern matching we are able to extract all the internal values of a node for use in the body of *f*. The variables *intN* and *subTreesN* are now bound to the values 2 and the empty list [], respectively. Another example is, let a=[1, 2, 3] be a list and (hd:tl) = a, then hd = 1 and tl = [2, 3]. One can achieve the same result by using the functions head and tail; hd = head a and tl = tail a. The variables, *hd* and *tl*, in pattern matching "(hd:tl) = a" are therefore often called the head and tail of list a. The universal pattern "_" matches every thing.

A way to create local declarations in Haskell is to use the *where* clause. The *where* let us put local declarations after the actual expression of the function.

```
factorialString n = "The factorial of  "++ show n ++" is " ++ fac
                    where
                            fac = factorial n
```

The above function takes a number and returns a string telling what the factorial of its argument is. We see that the string is constructed using the variable *fac*. This variable is declared in the *where* clause after it is used. The function show is a built in function that turns its argument into a string. The operator "++" is an infix operator that concatenates lists. A string in Haskell is just a list of characters.

1.3. Description of the remaining chapters

Other projects that have a close relation with our work are presented in chapter 2. We present projects that create and traverse syntax trees, that match sequences and trees and also that have a similar task as ours, but that consider other programming languages.

The major work of creating an AST comparison tool is described in the three major chapters, chapter 3, 4 and 5. In chapter 3 we discuss the problem of comparing Java programs. Java is a comprehensive general purpose, object-oriented programming language. There are many factors to consider when comparing Java programs. We present a short description of the language Java and the key areas of Java that we focus our thesis on. Furthermore we discuss other comparison tools, such as line-based tools. There is no tool to this date that identifies the syntactic differences between two Java programs, while also taking partial knowledge of Java semantic into account. It is therefore unfair to compare the tools that are presented in chapter 3 with our tool, since they originally have different purposes of use. Finally we give examples of comparisons, or rather example of good results of comparisons. And we suggest the best difference report based on the viewpoint of programmers.

We present algorithms for finding the longest common subsequences of two sequences in chapter 4. Our work rests heavily on the work of Yang [1]. We have extended the algorithms of Yang and also give new algorithms for supplement. Sequences can be viewed as trees of height two. The children of the root form the sequence. But we deal with much larger trees when comparing Java programs. In chapter 4 we also give algorithms for finding the largest common subtree.

In chapter 5 we implement the algorithms described in chapter 4. We transform the pseudo code into Haskell functions. The pseudo code is based on the imperative way of thinking and conflicts with Haskell's philosophy; some algorithms cannot be transformed to functions. We discuss this and also introduce a new attribute in the nodes of a tree. This new attribute is our first step towards adapting the functions to compare Java abstract syntax trees and not just the simple data structure that we created.

In chapter 6 we described the actual steps for comparing Java programs. We also describe the implementation of the Java comparing functions. The Java abstract syntax trees are complex and we have to make adaptations to the functions. We present a few comparison functions that are representative for the different techniques for comparing Java constructions, such as classes and methods.

In chapter 7 we test our tool up against other tools. We use two small files of an open source project and compared the iterations. The project is called Eclipse and it is developed by IBM. We compared the CVS version of two files and we generate reports using our tool and *diff*. The sizes of these reports are then plotted on a graph. Finally we conclude in chapter 8 with an overall statement and suggesting future works.

2. Related works

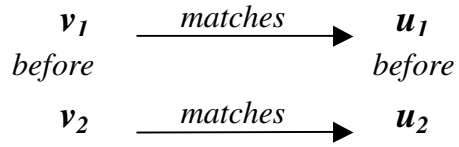
In this chapter we give on overview of works and projects closely related to our work. The most important source of support is Yang's paper [1]. Strafunski is a software bundle that we use to create and traverse syntax trees. We also present other systems similar to Strafunski, like *JJForester*. Furthermore we present works about syntax trees and sequence matching. Finally we present works about type matching, semantic differences and software merging.

2.1. Wuu Yang's paper

As mentioned earlier, this thesis rests heavily on the work of Yang [1]. In his paper Yang describes algorithms that exploit knowledge of the programming language's grammar. In his case it is the programming language C. The algorithms can point out the differences between two programs in a more accurate manner than lined-based tools, since line-based tools are intended for comparing ordinary text, not software code.

The two programs to be compared are first transformed into parse trees, or abstract syntax trees in modern terminology. Each node in these trees represents either a token or a non-terminal. A token is for example a variable name. Non-terminals denote a substructure, a composition of tokens and non-terminals such as expressions. Yang also gives five guidelines for building the tree representation. These guidelines are to ensure that the trees reflect the syntactic structure and the hierarchical structure of the programs, and to ensure that the size of the trees is kept to the minimum.

The algorithm matches the nodes of the two trees by using a dynamic programming scheme. A match is found between two nodes, one from each tree, when they contain identical symbols, their parents match each other and the order between siblings is respected. Meaning that if v_1 matches u_1 and v_2 matches u_2, and v_1 comes before v_2, then u_1 must also comes before u_2.



Yang's goal is to find the minimum syntactic distance between the programs. This is the same problem to finding the maximum syntactic similarity. When dealing with parse trees, this means that the goal is to find the largest common subtree. Any nodes in the original trees that are not represented in the largest common subtree are considered as a change, hence a difference to report. Yang considers the problem of matching trees as a generalization of the problem of matching sequences. Therefore Yang describes an algorithm for matching two sequences, with the aim of finding the longest common subsequence, before describing the algorithm for matching trees.

Yang also describes an algorithm that not only deals with matching nodes, but also comparable nodes. Assume that we want to compare the statement of the *while* loop against the statements in the *for* loop in the example below. The nodes that represent the root of the *while* loop and the root of the *for* loop are not identical. But to be able to compare the statement, their parent nodes must be identical. Yang overcome this problem by postulating that root of loops are comparable, although they are not identical.

```
while (w > 0) {          for ( i = 1; i < 1; i++) {
        x = 1;                   x = 1;
        y = 2;                   y = 2;
        z = 3;                   z = 3;
}                        }
```

This is an adaptation to the program language C. By implementing this adaptation, Yang is able to produce a more precise report.

2.2. Strafunski

Strafunski, [6] [9] [15], is a Haskell-centered software bundle. Its domain is generic programming and language processing. Strafunski supports generic traversal and is based on the notion of functional strategy. This is a set of generic functions that makes programs able to traverse a term, and subterms, of any types. And they are able to mix type-specific and uniform behavior.

A functional strategy helps programmers to construct more concise, robust and reusable programs since they can concentrate on the relevant constructors and deal with the other constructors generically. Functional strategies are composed as combinators and are therefore first-class generic functions. Strafunski provides the library StrategyLib containing generic programming combinators. We use Strafunski in our thesis to produce parse trees for Java programs, for traversing the trees and for extracting needed information. The package SDF2Haskell takes an SDF grammar[1], which is a set of syntax definitions, as input and generates Haskell data types for the abstract syntax trees. It is then possible to produce abstract syntax trees.

2.3. JJForester and other parsers and “tree builders”

JJForester [7] is a tool similar to Strafunski. Where the underlying language in Strafunski is Haskell, *JJForester* has Java. *JJForester* offers a library of reusable visitors that can be combined in many different ways to form new visitors and the result is full traversal control [6]. With *JJForester* one is able to traverse a abstract syntax tree and specify actions for a

[1] http://catamaran.labs.cs.uu.nl/twiki/pt/bin/view/Sdf/WebHome

limited number of nodes, nodes with a certain constructor or nodes that share a specific attribute. This is similar to Strafunski's type combinators for generic traversal [9].

JJForester combines advanced language processing technology in the *ASF+SDF Meta-Environment*[1], specially generalized LR parsing, with Java. The main domain of use for *JJForester* is component-based development of program analyses and transformations for languages of non-trivial size, [7].

In other words, *JJForester* is a parser and visitor generator for Java. Other parsers, tree builders and visitor generators for Java are e.g. *JavaCC*[2] and *The Java Tree Builder*[3]. The main differences between these tools and *JJForester* are the support for generalized LR parsing and the possibility for constructing new visitors from given ones.

2.4. Syntax trees and sequence matching

A number of algorithms exist for the problem of comparing sequences. Wu et al. [10] describe in their paper a sequence comparison algorithm whose running time is at most O(ND), where N is the sum of the length of the two sequences to be compared and D is the size of the minimum edit script between those sequences. The algorithm is best described as the problem of finding the shortest path from one point to another on a grid.

Charras and Lecroq [11] list 35 different algorithms for string matching. These algorithms use different approaches such as brut force, hashing and quick search.

Wang and Zhang [8] discuss in their paper the complexity of algorithms for four edit-based distance measures and they give an algorithm for one of them. They also establish a hierarchy among the four measures. The algorithm is based on the distance measure called "*Isolated-subtree mappings*". Basically it is an algorithm that finds the largest common subtree of two trees[4]. The relation between this algorithm and Yang's algorithm [1] (and our algorithms) is the fact that Yang' algorithm is a particular case of Wang's and Zhang's algorithm. Yang's algorithms give the upper-level nodes more weight than the children below and they demand that the root of the two trees must be identical. This is called a top-down mapping. Wang and Zhang in the other hand treats all the nodes equally and the subtree can be situated anywhere in the original trees, meaning that the root of the largest common subtree can match any nodes of the original trees.

Top-down mapping is the right choice when comparing programs because it considers the structure of programs. Consider the example of comparing the two program fragments in section 2.1. The roots of the loops are the *while* and *for* statements. A top-down mapping dictates that the roots must be identical, or in this case comparable, before the bodies of the loops are compared. This is reasonable because we must make sure that we comparing the same loops before comparing the rest. If the roots are not identical or incomparable, then the

[1] http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment
[2] https://javacc.dev.java.net/
[3] http://compilers.cs.ucla.edu/jtb/
[4] The algorithm is dependent on a constant that restrain the distance between the common substructures.

two fragments are not the same and we do not need to compare the bodies. An isolated-subtree mapping would have identified that the bodies of the two loops are identical.

2.5. Type matching

Jha, Palsberg and Zhao [12] present an algorithm for matching two recursive types. This algorithm is an O(n log n) time algorithm and this is done by reducing the problem to finding a size-stable partition of a graph. The problem in this case is determining whether two types are equivalent. The example in the paper shows four Java interfaces. The goal is to find out if one pair of interfaces is equivalent to the other pair. The interfaces in each pair are mutually recursive. The notion of equivalent in this case is that the interface names and method names do not matter, and neither do the order of methods and the order of formal parameters.

```
interface I1{                          interface I2 {
        float m1(I1 a, int b);                 J2 m3(float a);
        int m2 (I2 a);                         I1 m4(float a);
}                                      }

interface J1 {                         interface J2 {
        I1 n1(float a);                        int n3(J1 a);
        J2 n2(float a);                        float n4(int a, J2 b);
}                                      }
```

Above is an example of inputs to their algorithm. The algorithm then gives the following output:

$$I_1 = J_2$$
$$I_2 = J_1$$

$$I_1.m_1 = J_2.n_4$$
$$I_2.m_3 = I_2.m_4 = J_1.n_1 = J_1.n_2$$
$$I_1.m_2 = J_2.n_3$$

The types of interfaces I_1 and J_2 are equivalent. All their methods match. Furthermore we see that I_2 and J_1 are equivalent and all their methods match.

2.6. Semantic differences

The aim of Binkley et al. [13] is to capture the semantic differences of two programs. Their implementation takes two versions of a program and outputs an executable third program. The third program captures the semantic differences between the first two. The implementation is for the programming language C. A study of a collection of programs is done to investigate the time taken to compute the third program and the size of it relative to the size of newer

version of the input programs. The base of the implementation is the tools *CodeSurfer*[1] and *Cdiff* [1].

2.7. Software merging

Mens [14] does a stat-of-the-art survey of software merging where he gives an overview of merge techniques and describes the currently available tools that utilize the different techniques. On the subject of syntactic merging, Mens concludes that syntactic merging is more powerful than textual merging. Syntactic merging takes the syntax of the programs into account. There are two kinds of techniques of syntactic merging; abstract syntax trees and graphs. The problem of syntactic merging with abstract syntax trees as the underlying data structure is similar to the problem of finding the largest common subtree, and of shortest editing path in a graph.

Mens concludes furthermore that to be able to perform a syntactic merging, one must first compare the difference between two programs. Yang's paper [1] is mentioned in this particular chapter.

[1] http://www.grammatech.com/products/codesurfer/codesurfer.html.

3. The problem of comparing Java programs

In this chapter we give an overview of the challenges and opportunities when comparing Java[1] programs. Java is a complex language and has many aspects that have to be considered. We will give an introduction to the challenges that we face when comparing Java programs and discuss where line-based tools, like *diff*, and API-based tool, like *JDiff*, fail.

3.1. Java

Java is an object-oriented language. Object-oriented languages are popular because they can make it easier to reuse and adapt previously written software. Java comes with a comprehensive library. And developers often use packages in this library or reuse software written by others. When including an interface, the programmers must implement all its methods. And therefore changes might come in large bulks and affect the implementation a great deal.

Object-oriented languages regard objects as entities that encapsulate data and related operations, while procedural languages consist of procedures and data structures. According to Ghezzi [4], object-oriented languages are characterized by their support of four facilities:

- Abstract data type definitions
- Inheritance
- Inclusion polymorphism
- Dynamic binding of function calls to functions bodies

Java supports abstract data type by encapsulation using the class construct. Subclasses are defined in Java by extending an existing class. The subclass then inherits the implementation of its superclass. A class can only have one superclass, but it can inherit from several interfaces. By using the implements mechanism, Java supports the notion of multiple inheritances[2] [4]. Inclusion polymorphism allows the use of polymorphic variable that may refer to an object of a class or an object of any of its derived classes; any objects in Java may be assigned to a variable of type Object [4].

3.1.1. Class extension and interface implementation

All classes in Java are nodes in an inheritance and implementing tree. The root of this tree is the class Object. Every class therefore inherits the Object class, either directly or indirectly. The inheritance tree can be very complicated, since an interface may be implemented by one ore more classes and a class may be extended by several subclasses.

[1] http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
[2] But for interfaces, not classes.

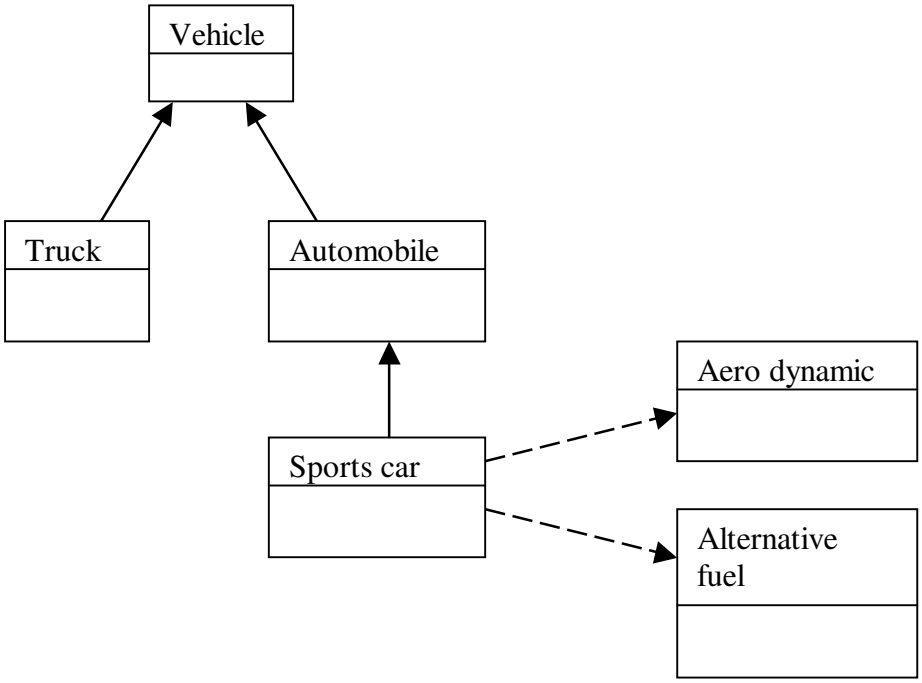


Figure 1. Inheritance tree

We see in Figure 1 that "*Vehicle*" has two subclasses, "*Truck*" and "*Automobile*". "*Sports car*" is a subclass of "*Automobile*". Class "*Sports car*" also implements the interfaces "*Aero dynamic*" and "*Alternative fu*el". This means that "*Sports car*" inherits the attributes of "*Automobile*" and "*Vehicle*", and it has to comply with the demands set by the interfaces.

A subclass inherits all the fields of its superclass and it may redefine the method of its superclass. The overriding method must have the exactly the same signature as the method being overridden. One cannot override *final* methods. *Static* methods are intended to perform class-specific rather than object-specific operations. The use of declaring methods as final is to protect important methods from being tampered with in subclasses. A class may also be declared to be final. In that case, the class may not be extended at all.

An interface may only contain constants and method declarations, but no implementations, unlike a class that can contain both. All methods declared in an interface are *public*. The implementation of the declared methods in the interface must be provided in any class that implements the interface.

3.2. Line-based and byte-based and API-based tools

Line-bases and byte-based comparison tools are designed to work with ordinary text or binary files. The Unix tool *diff* compares two files line by line, finds groups of lines that differ, and reports each group of differing lines. There are two ways of comparing two files. One is to

consider the files as text and compare them as mentioned earlier. The other is to consider the files as non-text and compare them byte-by-byte. Comparing byte-by-byte works best for checking whether two files are identical. Comparing text in this matter is not practical. For example, assume we have two identical files. Then we add an extra blank line in the beginning of one file, but not the other. The tool comparing byte-by-byte will report that every byte is different. Linux' *cmp* reports the position of the first differing byte.

The tool *diff* compares text files by finding large sequences of common lines and small groups of lines that differ. That way the report of differences is kept low. For practical purpose, this method is effective for textual comparison, especially if the changes are relatively small compared to the whole text.

Although *diff* has many options for formatting output, we get unsatisfactory reports when using *diff* on source code. The tool *diff* does not exploit the fact that source codes are bound to other syntactic structure than ordinary text. Therefore there are many things that line-based comparison tools do not solve satisfyingly. A simple example is the case of interchanging methods. Assume that two methods in the same class switch places from one version to the next. The tool *diff* cannot see that this change is really not a change at all to programmers or for the outcome of the program. Another example deals with object-oriented issues; consider when we implement interface *I* in class *C*. Class *C* has to implement automatic methods that it inherit from *I*. We may only be interested to knowing that *C* has implemented interface *I*. All the fields and methods that come along with that are not of interest. Example of this problem is described in section 3.4.5. The tool *diff* cannot see the difference between an inherited method and a declared method.

JDiff is a doclet[1] that produces an HTML report of differences between two Java programs. It compares packages, classes, fields and methods, but only the APIs, that is, only the method names and signatures. It does not look at the logic implementation of methods. This method of comparison is good and satisfyingly when the gap between programs is big and the details are not so important. E.g. when comparing differences between versions of the Java language or different version of large distributions.

3.3. Goals

Our goal with this project is to create a tool that compares two Java programs and reports changes regarding syntactic and contextual differences. The aim is not a tool ready for commercial production, but it has to improve the quality of the reports of differences. We will use the work of Yang [1] as base, but do adaptations towards Java. The reports of differences between two versions of a program will improve programmers' efficiency because they are able to share with each other the changes more accurately.

We do not in this version of the tool concern our selves with run-time efficiency, but rather focus on the quality of reports. Of course, if the algorithm takes a very long time to compute the results, one can question the usefulness of this tool compared with existing tools. Our main priority is still what the output is. We chose Java because it a very popular programming language and also because it supports the object-oriented paradigm. There are no tools that

[1] Doclet are a Java programs to specify the content and format of API documentation.

yet support the comparison of two programs in a way that takes grammatical structures and object-oriented constructs into account. As mentioned earlier, we have *JDiff* and *diff*. None of these compare programs in a satisfactory way regarding the mentioned criteria.

Java is a comprehensive programming language. We do not have the time to make a tool that considers every aspect of Java. In this version we concentrate on some of the aspects, mainly on object-oriented ones. We look at inheritance by implementations of interfaces. We also look at the method bodies, which *JDiff* does not do.

3.4. Examples of comparisons

We now give some examples of good comparisons, or rather good output from these comparisons. These examples illustrate both syntactical and contextual comparisons. Our tool does all of the comparison exactly as described in this section.

The best difference report is our view on what is important and usefull in a difference report when comparing two Java programs. The difference report is in a sense the least editing script from the first version to the second. By applying the changes in the report to the first version (version 1), we get the second version (version 2).

3.4.1. Implementing an interface

In this example we illustrate the comparison when a class implement an interface and we give the best difference report.

Version 1

```
interface FooPanel
{
        void meth1();
        void meth2();
}

class Foo
{
        int cf1;
        void meth3() {}
}
```

Version 2

```
interface FooPanel
{
        void meth1();
        void meth2();
}

class Foo implements FooPanel
{
        int cf1, cf2;
        public void meth1() {}
        public void meth2() {}
        void meth3() {}
        void meth4() {}
}
```

Java example 1. A class implementing an interface

In Java example 1 we have an interface *FooPanel*. It contains two method descriptions. Then we have class Foo, which has one field *cf1* and one method *meth3*. We see also what class *foo* looks like in version 2, after it implements interface *FooPanel*. Interface *FooPanel* remains the same. The best difference report for Java example 1 is:

- Class *Foo* implements *FooPanel*.
- Class *Foo* adds *int cf2*.
- Class *Foo* adds *meth4*() with the following definition: *{}*

Note that the adding of the methods *meth1* and *meth2* is not specified in the difference report. The first line and the semantics of Java interface implementations imply them, since they are inherited from *FooPanel*. Class *Foo* has added a field and a method.

A line-based tool would have told us that every line, except three, is changed in class *Foo* from version 1 to version 2. These tools will only recognize the brackets {} and the method *meth3* in both versions. They consider the other lines either as added or changed in some way. An API-based tool, such as *JDiff* will also report, in addition to the best difference report, that *Foo* has added the method *meth1* and *meth2*. It is not capable of extracting the important information since it does not take interface implementations into account.

The reports of a textual tool and an API-tool are also useful in this case, because the example is small and simple. But we can see now that it would not be satisfying in the long run. When programs get longer and more complicated, the report will not give us a clear overview of the changes.

3.4.2. Changes in an implemented interface

It is important to locate the source of the changes and report them where they belong. In this example we illustrate that changes in interfaces should only be reported as what they are; as changes in the interface and not as changes in the classes implementing them.

In Java example 2 we see that all the changes are done in the interface. Since class *Foo2* inherits from *FooPanel2*, it also inherits the changes. Class *Foo2* has only one method; *meth4*. It is not affected by any changes. The best difference report for Java example 2 is:

- Interface *FooPanel2* deletes *meth2*().
- Interface *FooPanel2* deletes *meth3*().
- Interface *FooPanel2* adds *meth5*().

We can see that the report only mentions interface *FooPanel2* and not class *Foo2*. Although the implementation of *Foo2* is altered because of the changes in *FooPanel2*, class *Foo2* itself has not been changed in any way. Method *meth4* is the only attribute that truly belongs to class *Foo2*. Everything else belongs to the interface, therefore it is natural to present the difference as changes in *FooPanel2*.

Version 1

```
interface FooPanel2
{
        void meth1();
        void meth2();
        void meth3();
}

class Foo2 implements FooPanel2
{
        public void meth1(){}
        public void meth2(){}
        public void meth3(){}
        void meth4(){}
}
```

Version 2

```
interface FooPanel2
{
        void meth1();
        void meth5();
}

class Foo2 implements FooPanel2
{
        public void meth1(){}
        void meth4(){}
        public meth5(){}
}
```

Java example 2. Changes in the interface

A line-based tool and an API-tool will report that many lines in both *Foo2* and *FooPanel2* are changed. As for the interface part, the report is equal to the ideal difference report. But it also contains a report of the changes in the class body of *Foo2*, which is identical to the report of the interface. Any report regarding class *Foo2* is unnecessary.

3.4.3. Removing an implemented interface

In this example we show that line-based and API-based tools fail because they lack knowledge of syntax and semantic of Java.

In Java example 3 we see that all the changes are done in class *Foo3*. The interface remains the same. Class *Foo3* in version 2 does not implement *FooPanel3* as it does in version 1. Therefore it does not inherit any fields or methods any more. The best difference report for Java example 3 is:

- Class *Foo3* removes interface *FooPanel3*.
- Class *Foo3* adds *meth1*().
- Class *Foo3* adds *meth2*().

We see that although the class body of *Foo3* has not been changed in any way, the report points out many differences. Class *Foo3* in version 1 defines only one method directly; *meth3*. Everything else it inherits from interface *FooPanel3*. Since it in version 2 no longer inherits from *FooPanel3*, therefore it must define the methods *meth1* and *meth2*.

Version 1

```
interface FooPanel3
{
        void meth1();
        void meth2();
}

class Foo3 implements FooPanel3
{
        public void meth1(){}
        public void meth2(){}
        void meth3(){}
}
```

Version 2

```
interface FooPanel3
{
        void meth1();
        void meth2();
}

class Foo3
{
        public void meth1(){}
        public void meth2(){}
        void meth3(){}
}
```

Java example 3. A class removing an interface

A line-based tool and an API-tool will only report that the line "*class Foo3 implements FooPanel3*" has changed to "*class Foo3*". Everything else has not been altered. This is an insufficient report. Class Foo3 has many importing changes that the tools will not point out.

Java example 3 illustrates the inadequacy of a line-based tool. Comparing programs line by line[1] is not an ideal way of comparison. In this example it fails, though the number of code lines is relatively small.

The signature of the methods of class *Foo3* in version 2 has not been changed in any way. *JDiff* do not care whether the methods are inherited or not. Its only concern is to report API differences. Therefore it only reports that the declaration of class *Foo3* is changed.

3.4.4. Changes in the method body

We will now show some examples that illustrate where tools such as *JDiff* are not sufficient at all. These examples compare method bodies. *JDiff* only compares at the API level and therefore do not detect any changes done to methods beyond their signature.

In Java example 4 the method body is changed. The two first lines have been switched and the result is different values of the field *a* when the method returns. In version 1, *meth1* returns value 1, while *meth1* in version 2 returns value 2. The best difference report for Java example 4 is to point out that the lines "*a=a+a*" and "*a++*" have been altered in some way. Both *diff* and our tool face the problem of choosing which line to report as the one been altered. This problem is addressed in section 4.2.4.

[1] The tool *diff* tries to group large sequences of matching lines together, called hunks. That way the report of difference is kept low.

Regardless of which one is chosen, the important is that the tool reports that a change has occurred. Line-based tools work well when comparing method bodies because they are relatively small and the structure is similar to ordinary text. Statements in a method body can be regarded as a sequence of lines where the order is significant. They report whenever a line is changed, moved or altered in any way, but not if the file is reformatted.

Version 1

```
class Foo4
{
        int a=0;

        int meth1()
        {
                a=a+a;
                a++;
                return a;
        }
}
```

Version 2

```
class Foo4
{
        int a=0;

        int met1()
        {
                a++;
                a=a+a;
                return a;
        }
}
```

Java example 4. Changes in method body

Our tool works in a similar way as line-based tools when comparing method bodies, but our tool exceeds line-based tools in some areas, such as white-space. See section 3.4.5. *JDiff* does not report anything in Java example 4.

3.4.5. Ordering and white-space

Here we have another example where line-based tools fail. Again, line-based tools fail because they lack knowledge of Java syntax.

Version 1

```
class Foo5
{
        void meth1(){}
        void meth2(){}
}
```

Version 2

```
class Foo5
{
        void meth2(){}
        void meth1(){}
}
```

Java example 5. The order of methods

In Java example 5 the order of the methods has changed. In Java the order of methods is not significant and the best difference report should not report anything in this case. *JDiff* does not report anything and neither do our tool. But a line-based tool will point out that either the line "*void meth1(){}*" or "*void meth2(){}*" has been altered. This information is unnecessary.

(i) *x = 1; y = 2;*

(ii)
x = 1;
y = 2;

White-space confuses line-based tools. Java does not consider white-space characters syntax and therefore ignores them[1]. The example above illustrates this. Version (i) and (ii) are in the eyes of Java identical or at least equivalent. Line-based tools see a difference here.

3.4.6. Changes of the parameter names

In this example we address the situation of where the parameter names changes from one version to the next. A formal parameter has the form "*parameter-modifier type parametername*". Formal parameters are given as a comma-separated list when declaring a method. The formal parameters can be viewed as initialized variable, and their scope is the method body. The order of the formal parameters affects the signature of the methods. Or to be more precise, the order of the type of the parameters affects the signature. The method name together with the parameter types forms the methods signature. We compare signatures when checking whether two methods are the same.

Version 1	***Version 2***
public void meth(int a, int b) {}	*public int meth(int c, int b) {}*

Java example 6. Changes of the parameter names

The signatures of the two methods are identical. The difference is that the name of the first parameter has changed. The best report of differences for Java example 6 is:

- Parameter *a* of method *meth(int, int)* changes to parameter c.

A lined-based tool like *diff* sees only that the line differ from version 1 to version 2, while a API tool like *JDiff*, does not concern about this change because it does not affect the signature of the method.

[1] Only whit-space in syntactic correct code.

4. Fundamental comparison algorithms

In this chapter we present and introduce algorithms for comparing two programs. These algorithms are the foundation that our tool is built upon. As mentioned, the problem of finding the differences between two programs is equivalent with the problem of finding the shortest editing script from one version to the other. And finding the shortest editing script and finding the largest similarity are two sides of a problem. We present therefore algorithms for finding the largest similarities.

Our approach for comparing programs is to transform them into abstract syntax trees and then compare them. The largest similarity between two trees is the largest common subtree. In our case, we can view sequences as trees with height two. We give therefore algorithms for comparing sequences, which is algorithms for finding the longest common subsequence. And we give also algorithms for finding the largest common subtree.

At this stage, we work only with sequences of integers and nodes with integers. Integers are preferred because it is easy to compare them and they make simple data structures. We can then focus on the fundamentals of the algorithms.

4.1. Notation

In this section we will explain the most important notations and convention that we use later in this thesis. Some minor notations will be explained later where it is necessary and appropriate.

4.1.1. Sequences

Sequences are an ordered set of elements. We use A and B as variable names for sequences. They will have a number as suffix, i.e. A1, A2, B3 etc., for ease of use and distinction. An example of a sequence of integers is A1=(4, 7). The order of the elements is significant. I.e. (4, 7) is not the same sequence as (7, 4). The first element in A1 is “4”, and has index 1, and the second element is “7”, which has index 2. The indexes always start at 1 and it increments by one for each element. For sequence A, |A| denotes the length of A.

Another way of addressing elements in sequences is to use the notation A<i> where i is the index of sequence A, i.e. A1<2> = 7. When the notation is used with two or more indexes, then we are addressing a subsequence, i.e. A1<1, 2> = (4, 7). Furthermore we have A1<1,…, i>, which is a subsequence with every element in A1 with index from 1 to i. We will use i as index for sequence A and j for B.

A subsequence is an ordered set of one or more elements of the original sequence. If we have sequence A, and sequence Q is a subsequence of A, then Q is derived by deleting some elements of A. We will use Q as variable names for subsequences. They will have suffix for distinction. Sequence Q is a common subsequence of A and B if Q is a subsequence of both A and B.

Subsequence Q is the longest common subsequence if the length of Q is the longest among all the common subsequences of A and B. If we have A1=(1, 2, 3) and B1=(1, 2) then the longest common sequence Q1=(1, 2). And if we have A2=(1, 2, 3, 4) and B2=(1, 2, 4), then the longest common subsequence Q2=(1, 2, 4). Q2 is called a broken subsequence because the pairs of elements do not follow in order, but skip one or more elements. We see that element A2<3> is skipped because it does not match any element in B2.

4.1.2. Matrices

A matrix is a two-dimension grid. In our case, the matrices contain elements of integers. The elements are integers because they denote the length of the longest common subsequence between two sequences. We use the letter M to name a matrix and we add a suffix to distinguish between them.

Matrix M is a result of Yang's sequence matching algorithm, described in Algorithm 1. The size of the matrix depends on the length of the sequences that are given to the algorithm as arguments. Examples of matrix M are found in Table 1. We see in the table that the length of sequences A1 and B1 are both four. It is natural to think that the sequence will produce a matrix of size 4×4, but the size of the matrix M1 is 5×5. This is because the definition of Yang's matrix requires that an extra row and column be added. They are necessary for computing the rest of the elements, or rather to get a compact formulation of the algorithm.

For ease of discussion later, we introduce some conventions for addressing M.

1. M means the whole matrix.
2. M[i, j] is the element found at indexes i and j. Index j is the index for the rows and i is the index for the columns.
3. Index j is also the index of sequence B and i is the index of A. This is because of the close relation between the sequences and the matrix. See Table 1.
4. The elements in row i=0 and column j=0 are zero by definition.
5. We often think of matrix M as a long sequence by imagining that the rows in matrix M are lined up one after another, and form a linked sequence. The end of the first row is linked to the beginning of the next etc. See Figure 2. Note that this is not a representation of data structure, but rather a cognitive concept.
6. When we talk about the beginning and end of matrix M, we think of the beginning and end of the imaginary and flattened matrix. See Figure 2.

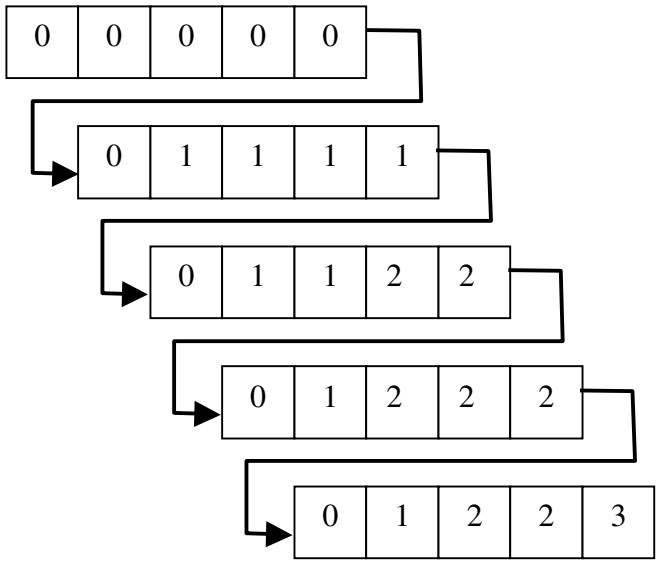


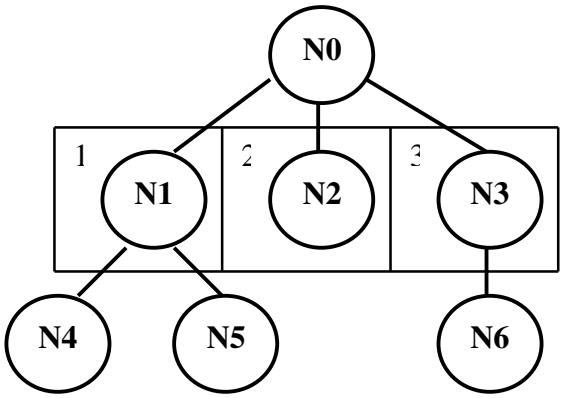
Figure 2. Flattened M

4.1.3. Trees

A tree is a way of storing data. It is a structure that contains elements called nodes. We will use the N when addressing a single node. A number for ease of use and distinction will suffix them. Every node has a value. The values are the data that we want to store. The tree structure helps us store the data.

Figure 3. Example of a tree

Following are some convention we use for addressing trees:



1. T means the whole tree.
2. The root-node, or just root is the top node in a tree. It does not have any parent. The node N0 is the root in Figure 3.
3. A leaf node is a node that does not have any children. The nodes N4, N5, N2 and N6 are examples of leaf nodes.

4. A branch node is a node that is not a leaf node or root. The nodes N1, N2 and N3 are examples of branch nodes.
5. If tree subT is a subtree of tree T, then subT is derived by eliminating some nodes of T. The root of tree T is also the root of subT and the parent-child relationships between the nodes in subT are the same in as in T.
6. The tree subT is a common subtree of the trees T1 and T2 if subT is subtree of both T1 and T2.
7. The largest common subtree subT of T1 and T2 is one with the maximum number nodes among all the subtrees of T1 and T2.
8. Any node can be viewed as the root of its subtree. In Figure 3, node N1 is the root of the subtree that contains the nodes N1, N4 and N5.
9. The children of a node are that node's subnodes. In Figure 3 are N1, N2 and N3 the subnodes of N0.
10. Subnodes can be viewed as a sequence. In Figure 3, we see that N1, N2 and N3 form a sequence with index from one to three.
11. One can address a subtree by using the notion *i*th subtree of N, where i is the index of the sequence formed by the subnodes of N. In Figure 3, N1 is the first subtree of N0. And further, we have N5, which is the *2*th subtree of subtree N1.
12. We have node Nx and Ny and the sequences A and B are the sequences containing Nx' and Ny's subnodes respectively. Sequence matching with the nodes Nx and Ny as arguments is the same as sequence matching with the sequences A and B as arguments.
13. We have a tree T, then |T| denotes the number of nodes T has.
14. The notion |subtrees of N| or |subnodes of N| denote the numbers of children N has. In Figure 3 we see that |subtrees of N0| is three.

4.2. Comparing sequences

Follow are descriptions of algorithms for comparing sequences. We present Yang's algorithm for comparing sequences and our algorithm for finding the longest common subsequence. We also give an algorithm that combines the two mentioned algorithms. And last we discuss ways of reducing the size of the matrices.

4.2.1. Sequence matching algorithm

The algorithm for finding the length of the longest common subsequence for two sequences is described by Yang [1]. Examples of results of the algorithm are shown in Table 1. Although this algorithm computes the length, it does not point out the actual longest common subsequence. It is necessary to identify this subsequence and we also give an algorithm for this.

Algorithm: Sequence matching (A, B)
m := |*A*|
n := |*B*|
Initialization. *M [i, 0] := 0 for i = 0,…, m.*
M [0, j] := 0 for j = 0,…, n.

for *i := 1* ***to*** *m* ***do***
for *j := 1* ***to*** *n* ***do***
M [i, j] := max (M[i, j-1], M[i-1, j], M[i-1, j-1] + W[i, j])
where *W[i, j] := 1 if* $A_i = B_j$ *and W[i, j] := 0 otherwise*
end
end
return(M)

Algorithm 1. Sequence matching

Variable m and n in the algorithm **Sequence matching** denote the length of sequence A and B. They are necessary for computing the matrix M. Element M[m, n] denotes the length of the longest common subsequence when comparing sequences A and B. The algorithm, as describes by Yang [1] returns only this element (M[m, n]) when the algorithm terminated, not the whole matrix (M) as we do in the algorithm **Sequence matching**. But we need the whole matrix in algorithm **Longest common subsequence** to be able to find the longest common subsequence.

B = (2, 3, 4, 5)

M =

A =	**i \ j**	**0**	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
(1,	**1**	0	0	0	0	0
2,	**2**	0	1	1	1	1
3,	**3**	0	1	2	2	2
4)	**4**	0	1	2	3	3

Table 1. Example of result of Wuu Yang Sequence Matching algorithm

Matrix M in Table 1 is a result of algorithm **Sequence matching** with A and B as arguments. It is to be interpreted as follows. The entry M[i, j] denotes the length of a common subsequence of the two prefixes A<1,…, i> and B<1, …, j>. Therefore M[1, 1] denotes the length of a common sequence of A<1> and B<1>. This is zero in matrix M because the two elements A<1> and B<1> are not identical. Furthermore we have the entry M[2, 1], which tells us that the sequence A<1, 2> and the element B<1>, has one common element (A<2> = B<1> = “2”). The entry M[3, 2] tells us that there are two common elements between

sequences A<1,…,3> and B<1,…2> (A<2> = B<1> and A<3> = B<2>). Finally M[4, 4] (which is the last element in matrix M, see point 6 in section 4.1) tells us that there are three common elements when comparing A and B, which is the length of the longest common subsequence.

4.2.2. Observations regarding matrix M

Understanding the patterns that emerges from the matrices is the key to identifying the longest common subsequence. Following is a list of rules regarding matrix M.

Let matrix M be the product of algorithm **Sequence matching** with the sequences A and B as arguments, and m=|A|, n=|B|. And the element M[i, j] is a match if A<i> = B<j>.

Rule 1. The element M[m, n] always denotes the length of the longest common subsequence of sequences A and B.

Rule 2. When traversing M, a match is found at M[i, j] if and only if the value of M[i, j] is exactly one larger than each of the values at M[i-1, j], M[i, j-1] and M[i-1, j-1], for $i \in [1…m]$ and $j \in [1…n]$.

Rule 3. If a match is identified at M[i, j] then the next match does not occur until at least M[i+1, j+1], for $i \in [1 … m-1]$ and $j \in [1 … n-1]$. If we consider M as flattened (see point 4 in section 4.1), then there are no match between M[i, j] and at least M[i+1,j+1].

Rule 4. If there exists only one instance of the longest common subsequence Q, and Q is not broken (see section 4.1.1), then as a result of rule 3, Q=(M[i, j], M[i+1, j+1], M[i+2, j+2]… M[i+(c-1), j+(c-1)]), where M[i, j] is the first match and c=M[m, n], for $i \in [1 …m]$ and $j \in [1 …n]$.

Rule 5. If the sequences contain multiple common subsequences of the same maximum length, see Table 2, then the path from one matching element to the next matching element branches out to different path. Each branch represents a common subsequence. All the above rules, except no 4, still apply. See Table 2.

4.2.3. Identifying the longest common subsequence

The algorithm for identifying the longest common subsequence (Algorithm 2) first applies **Sequence matching** algorithm to its arguments. The result is returned as matrix M. The variable C, which denotes the length of the current longest common subsequence, is initially set to zero. It then finds the subsequence by analyzing the matrix M. The algorithm does an exhaustive search through the matrix, starting with element M[1, 1]. Every time it finds a value in M[i, j] that is greater than C, it ads an element to Q by looking up in sequence A using index i. When an entry in matrix M is greater than C, it means that Yang's algorithm has found a match in the two sequences A and B. The value C remains unchanged until another match is found.

When applying the examples of Table 1, sequence A1 and B1, to algorithm **Longest common subsequence**, we get the longest common subsequence Q1=(2, 3, 4). And we get Q2=(3, 4, 5) of the sequences A2 and B2. Note, if we apply the sequences (1, 2) and (2, 1) to the algorithm **Longest common subsequence**, the result is the common sequence (1), although the common

sequence (2) is also an adequate solution. The problem of multiple solutions is addressed in section 4.2.4.

```
Algorithm: Longest common subsequence (A, B).
M := Sequence matching(A, B).
C := 0 , length of the current longest subsequence.
m := |A|.
n := |B|.
Q := sequence of length M[m, n]. Contains the longest common subsequence when
the algorithm terminates.
k := 0, index for Q.

for i := 1 to m
        for j := 1 to n
                if M[i, j] > C then
                        Q[k] := A[i]
                        C := M[i, j]
                        k := k+1
                endif
        end
end
return ( Q )
```

Algorithm 2. Longest common subsequence

4.2.4. Example with multiple solutions

The entry M[m, n] always tells us the length of the longest common subsequence, but there are times when it does not tells us which one. Sometimes two sequences contain several common sequences of the same length. Example, A = (1, 10, 2, 20) and B = (1, 2, 10, 20) have two common subsequences, {1, 10, 2} and {1, 2, 20}. Both are equally valid and have the same length.

It is possible to retrieve every possible solution from matrix M. As mentioned earlier, the characteristics of a match, for example in M[i, j], is that it is exactly one greater than M[i-1, j], M[i, j-1] and M[i-1, j-1], see Table 1. In the example in Table 1 the path of matching elements starts at entry M[2, 1] and then continues southwest. If m=|A|, n=|B| and the first common entry is M[i, j] and we know that the sequence of common subsequences Q is not broken, then we know that Q= (M[i, j], M[i+1, j+1], …, M[i+m, j+n]). But when there are several solutions, the path is not straight, but it will branch out into equally numbers of path as there are solutions, see Table 2. This makes it harder for us to identify the common entries, but the rules stated in section 4.2.2 still apply. As seen in Table 2, every common entry is one greater than the previous entry.

A := (1, 10, 2, 20)
B:= (1, 2, 10, 20)

M := Sequence matching (A, B)

M

i \ j	0	1	2	3	4
0	0	0	0	0	0
1	0	**1**	1	1	1
2	0	1	1	**2**	2
3	0	1	**2**	2	2
4	0	1	2	2	**3**

Table 2. Example with multiple common subsequences

The entries that mark the common subsequence in the example in Table 2 are M[1, 1], M[2,3] or M[3, 2], and finally M[4, 4].

The algorithm for identifying the longest common subsequence does not in this thesis compute all possible solutions. It will only find one solution and always the first branch. The first branch is the path of where the index of the matching elements is the lowest. When choosing between two entries, the algorithm always favors the on with lowest index, as in the prior element in a flattened M, see point 6 in section 4.1.2. I.e. we have entry $M_E[i_E, j_E]$ and $M_F[i_F, j_F]$. M_E is prior to M_F if $i_E < j_E$. If $i_E = j_F$, then M_E is prior if $i_E<j_F$.

4.2.5. Combining the matching and identifying algorithms

We see that by applying two sequences to algorithm **Sequence matching** and then the result of this to algorithm **Longest common subsequence** we get the desirable results. Note that algorithm **Longest common subsequence** only needs |A|, |B| and M. But it is an inefficient way of computing the longest common subsequence. The matrix M is first produced and then traversed. A dynamic programming scheme will remove unnecessary steps and will improve the efficiency. We suggest that the longest common subsequence is identified at the same time as the matrix M is constructed. It is then not necessary to go through the matrix more than once. We cannot eliminate the matrix entirely because it ensures the finding of a maximum matching and every solution for this.

A negative aspect of our suggestion is the loss of multiple solutions. Although our algorithms presented so far do not report every adequate solution, it does not totally exclude the possibility of finding the other solutions. The information for identifying those solutions still lies in the matrix that is produced. But with the combined algorithm we now present there will be only one possible solution because algorithm **Combining the matching and identifying algorithm** discards the matrix M after the longest common subsequence is identified and therefore no other solution can be identified. The choice, of which solution to pick, is the same as describe in section 4.2.4.

```
Algorithm: Combined algorithm (A, B)
m := |A|
n := |B|
Initialization.  M[i, 0] := 0 for i = 0, …, m.
                 M[0, j] := 0 for j = 0, …, n.
C := 0, length of current longest common subsequence.
Q := sequence of length M[m, n]. Will eventually contain the longest common
subsequence.
k := 0, index for Q.

for i := 1 to m
        for j := 1 to n
                M[i, j] := max ( M[i, j-1], M[i-1, j], M[i-1, j-1] + W[i, j])
                        where W[i, j] := 1 if A_i = B_j and W[i, j] := 0 otherwise

                If  M[I, j] > C then
                        Q[k] := A[i]
                        C := M[i, j]
                        k := k +1
                endif
        end
end
return (Q)
```

Algorithm 3. Combining the matching and identifying algorithm

4.2.6. Reducing the size of the matrix

The algorithm for identifying the subsequence searches through every entry of the matrix M. It is not necessary and it is of course time consuming, especially when we are dealing with full-scale programs and not small and trivial sequences. There are however a few ways that would improve the efficiency.

The first way is to start the traversing at element M[1,1] because we know that there is no match in the first row and column. By definition the first row and column consist of zeros. This saves us from traversing the whole first row.

The second way is once we have found the starting point, the first element in the matrix that denote a match, we know that the next match can the earliest be found at the next row and one column, see section 4.2.2. This means that when we find an element, we can ignore parts of the matrix because the rest is irrelevant. If we have a match at element M[i, j], then we cut off the rows with index i $\in$ [0…i] and the columns with index j $\in$ [0…j]. And continue the traversing with element M[i+1, j+1].

This method is efficient and usable because we know how to avoid cutting off the relevant elements. However when we are dealing with multiple solutions, we will only retrieve one solution and always the first one, as in the first one in the flattened M, see section 4.1.2.

4.3. Comparing abstract syntax trees

In this section we give algorithms for comparing abstract syntax trees. We present a modified version of Yang's algorithm and give an algorithm for identifying the largest common subtree. Yang's algorithm assumes that the order of the subtrees of a node is significant, but we also want the order to be insignificant. We give therefore algorithms that handles both ordered and unordered nodes.

4.3.1. Simple tree matching algorithm

The algorithm presented here computes the number of nodes of the largest common subtree of two trees. First of all it checks whether the roots of the trees T1 and T2 matches. If they do not, the algorithm returns 0, as there are no common nodes between T1 and T2.
If they match, the algorithm recursively finds the number of pairs in a maximum matching between the two trees.

Algorithm: Simple tree matching (T1, T2)
if *the roots of the two trees A and B contain distinct symbols* ***then return*** *(0).*
m := the number of first-level subtrees of T1.
n:= the number of first-level subtrees of T2.
Initialization. M[i, 0] := 0 for i= 0, …, m.
M[0, j] := 0 for j=0, …, n.
for *i := 1* ***to*** *m* ***do***
for *j := 1* ***to*** *n* ***do***
M[i, j] := max (M[i, j-1], M[i-1, j], M[i-1, j-1]+W[i, j])
Where W[i, j] = Simple tree macthing ($T1_i$, $T2_j$)
Where $T1_i$ and $T2_j$ are the ***i****th and* ***j****th first-level subtrees of T1 and T2, respectively.*
od
od
return *(M[m, n] + 1).*

Algorithm 4. Simple tree matching

Matrix M computed by algorithm **Simple tree matching** is the result of matching the two trees. The entry M[m, n] denotes the number of the maximum matching pairs. Note that M[m, n] is added 1 when the algorithm terminates. This is done to account for the fact that the roots of the two trees match. Matrix W in algorithm **Simple tree matching** is nothing more than a matrix M, but it is a result of a recursive call with two subtrees as arguments. Matrix M is therefore computed from results of several W matrices. And W matrices are again dependent

on results from W matrices generated from lower subtrees and so on. The result of all W matrices is then passed up until they reach the top matrix, which is M. Algorithm **Simple tree matching** is best illustrated by an example, see section 4.3.2.

4.3.2. Example with the Simple tree matching algorithm

We have two trees T1 and T2 as shown in Figure 4. Each node contains only one number and they are named for ease of reference. A pair of nodes matches if they contain identical number and their parents match. In the case of the roots, it is only necessary for the numbers to be identical for them to match.

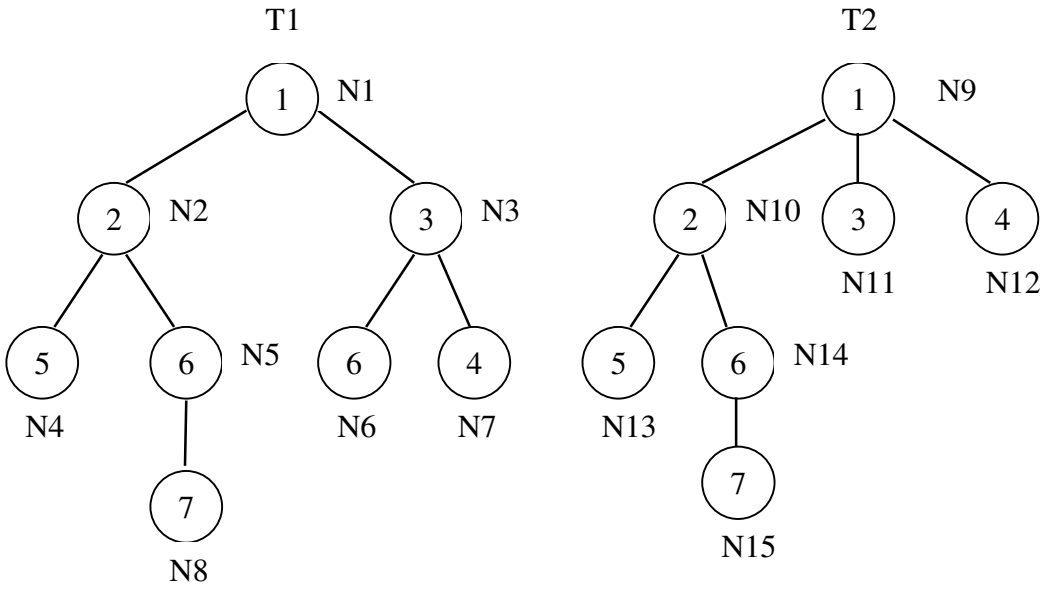


Figure 4. Examples of simple trees

After checking the roots, the algorithm then recursively calls itself with the two first subtrees of N1 and N9 as argument. The result is stored as matrix W. Matrix M2 in Table 4 is in fact one of the matrix W in matrix M1. Matrix M2 tells us that there are four matching pairs; {N2, N10}, {N4, N13}, {N5, N14} and {N8, N15}. But before the algorithm could yield the matrix M2, it had to compare each subtree of N2 (N4 and N5) against each subtree of N10 (N13, and N14). And the algorithm keeps on going to lower levels until there is no match or until it reaches a leaf node. Once algorithm, with N2 and N10 as arguments, terminates the value 4 is stored in M1[1, 1]. The algorithm then continues with comparing N2 and N11, which yields the value 0 because they don't have identical characters. The algorithm continues with comparing the pairs {N2, N12}, {N3, N10}, {N3, N11} and {N3, N12}.

When the algorithm terminates, M1[2, 3] is 5 and the algorithm will return 6, for the six pairs in a maximum matching. We can see that in this case it is the pairs {N1, N9}, {N2, N10}, {N4, N13}, {N5, N14}, {N8, N15} and {N3, N11}. Note that N14 has the same character as

N5 and N6. The same goes for N12 and N7. But it is only N5 and N14 that is a match since they are the only ones that the parents also match.

M1 = Simple tree matching (T1, T2)

M1

	0	1(N10)	2(N10-N11)	3(N10-N12)
0	0	0	0	0
1(N2)	0	4	4	4
2(N2-N3)	0	4	5	5

Table 3. Matrix M when applying algorithm on T1 and T1

As mentioned before, the subtrees N2 and N10 will yield the value 4 and we can see that it is situated at position M1[1, 1] in Table 3. When the algorithm recursively calls itself with N2 and N10, the following matrix (Table 4) is then generated.

M2=Simple tree matching (N2, N10)

M2

	0	1(N13)	2(N13-N14)
0	0	0	0
1(N4)	0	1	1
2(N4-N5)	0	1	3

Table 4 . Matrix M2(W) when applaying algorithm on N2 and N10

We can see that M2[2, 2] in Table 4 is 3. This means that there are three matching pairs under N2 and N10. And if we account for the fact that N2 and N10 also match, then the total maximum matching for the subtrees N2 and N10 is 4, which is the value passed on to element M1[1,1] in Table 3.

This algorithm produces large amount of matrices, but only returns the number of the largest common subtree. This cause some challenges for Algorithm 5. It has to reproduce most of these matrices to be able to identify the largest common subtree. The reproduction is the topic for discussion in section 4.3.6.

4.3.3. Identifying the largest common subtree.

As with the sequence-matching algorithm, **Simple tree matching** does not point out the common nodes. We give an algorithm to compute this.

Once again we use the matrix M to extract information so that we can construct the largest common subtree. But there are now not just one matrix M. We have now several matrices at different levels. The top-level matrix is dependent on the matrices at the lower levels. See

section 4.3.2. An exhaustive search through the top-level matrix M finds the matching pairs of two sequences of the root's subnodes. Whenever a match is found, the algorithm then does another exhaustive search. It calls itself recursively with the matching nodes as arguments, or more exactly the subtrees where the matching nodes are the roots. Each time a match is found the algorithm remembers the position of the nodes and uses them later to construct the largest common subtree.

```
Algorithm: Largest common subtree (T1, T2)
if T1 or T2 is empty then return empty tree.
M := Simple tree matching (T1, T2)
m := |subtrees of T1's root|
n:= |subtrees of T2's root|
if M[m, n] = 0 then return empty tree
commonSubTrees := Empty list. Will contain the list of common subtrees.
 subT1i := the ith subtree of  T1's root
subT2j := the jth subtree of  T2's root

for i:= 1 to m do
        for j:= 1 to n do
                if M[i, j] > M[i-1, j] then
                        commonSubTrees := commonSubTrees + Largest common subtree
                (subT1i, subT2j)
                endif
        od
od
return (T1's root with commonSubTrees)
```

Algorithm 5. Largest common subtree

The first algorithm **Largest common subtree** does is to check whether T1 or T2 are empty. There is no point of going further if at least one of them is empty. Then it passes its arguments to the algorithm **Simple tree matching**. The result is matrix M. The list *commonSubTrees* is initially empty. It will eventually contain the list of common subtrees of T1's root and T2's root. The algorithm then traverse matrix M. When a match is found, that is $subT1_i$'s root is identical to $subT2_j$'s root. The algorithm then calls itself recursively with those nodes. The result is the common subtree of $subT1_i$'s root and $subT2_j$'s root it is joined together with the list *commonSubTrees*. The operator "+" is an infix list concatenation operator.

When the algorithm finishes with traversing the lower-level matrix M, then it continues traversing the top-level matrix M. At the end, when every element is checked and every matrix is traversed, it joins T1's root with the list of common subtrees of T1's root and T2's root. This forms the largest common subtree of T1 and T2.

4.3.4. Identifying a match

Although the matrices for tree comparison look like the ones for comparing sequences, there are some important differences. First of all we see that the entries make larger leaps than before. With sequences, the entries are not able to increase with more than one at a time. See Rule 2 in section 4.2.2. This is because with sequences, one can at most find one more match at any given step of the matching process. But with trees the entries are not bound by that constraint. We see, e.g. in Table 3 and Table 4 that the entries increase with four at the most. Element M1[1, 1]in Table 3 makes a leap from value 0 (in the previous elements, see section 4.1.2) to the value 4. The reason is that element M1[1, 1] denote the number of the largest common subtree of the subtrees where N2 and N10 are the roots.

Second, the sequences produced only one matrix. With trees, there is one matrix for every comparison, because **Simple tree matching** algorithm needs them to compute. So when searching for matches, we must go through everyone to be make that we get all possible common nodes.

4.3.5. Observation regarding matrix M when comparing trees

In section 4.2.2 we set out a set of rules for matrix M for sequence matching. They still apply to tree matching with a few modifications. Let A and B be sequences of subnodes of T1 and T2's roots, and M is the result of simple tree matching with A and B as argument. The modifications are as follow:

Rule 1. We have tree T1 and T2 and m=|subtrees of T1's root|, n=|subtrees of T2's root|. Then M[m,n] always denotes the number of nodes in the largest common subtree of T1 and T2.

Rule 2. A match is found at M[i, j] if and only if the value of M[i, j] is larger than the values at M[i-1, j], M[i, j-1] and M[i-1, j-1]. And M[i-1, j] = M[i, j-1] = M[i-1, j-1].

A new for trees:

Rule 6. When the value of a match at M[i, j] is k larger than M[i-1, j], M[i, j-1] and M[i-1, j-1], it means that there are k matching nodes when comparing the subnodes of the sequences A<1…i> and B<1…j>. The nodes at position A<i> and B<j> are a match, and the remaining k-1 matches can only be identified by analyzing the matrix generated when giving the before mentioned nodes as argument to **Simple tree matching** algorithm.

The other rules apply unmodified to trees.

4.3.6. Discussion

The algorithm **Largest common subtree** produces a large number of matrices because new ones are generated every time the algorithm calls itself recursively. They are not all necessary even though they are all unique, in the sense that the inputs are different every time. The matrices are produced, and then thrown away and then the exact same ones are some times reproduced. This is a waste of space and time. We illustrate this further with an example. We have two trees T1 and T2 as seen in Figure 5. They consist of three levels, meaning three

generation of nodes, each level having only one node. The steps for computing the largest common subtree are described in Figure 6.

In step 1 we have concluded that the roots of the trees are identical, therefore we apply the **Simple tree matching** algorithm to the roots. The result is a matrix M0. But we need matrix M1 and matrix M2 to be able to compute matrix M0. The result of matrix M2 is passed up to matrix M1, which in turn passes its result to matrix M0. Now we have finished computing and are ready for identifying the common subtree. Keep in mind that **Simple tree matching** does not point out the nodes that are common. It simply tells the number of pairs in a maximum matching between two trees.

In step 2 we traverse matrix M0 to find the common nodes. We do an exhaustive search through the matrix and every time we find a match, we move on to the next step. In this case there are only one element to traverse and it is a match. In step 3 we have identified the pair N2 and N5 as a match. We must then reproduce matrix M1 so we can identify possible common nodes in level 2. And as before, we have to produce matrix M2 before we can compute matrix M1. Once we have computed matrix M1, then it is ready for traversing. This is done in step 4.

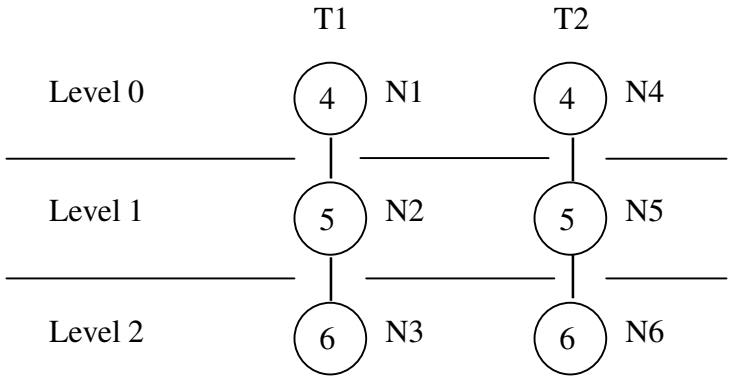


Figure 5. Trees with only one node in each level

In step 5 we have identified the nodes N3 and N6. We produce again matrix M2. Since at least one of them is a leaf node, we do not need to produce any lower-level matrices. In step 6 we traverse matrix M2. But matrix M2 does not indicate any more common nodes, therefore the algorithms terminates here.

If we have several more nodes for example in level 1, then step 3 to step 6 must be repeated with the number of matched found in M1. This means that additional matrices in level 1 and 2 must be created.

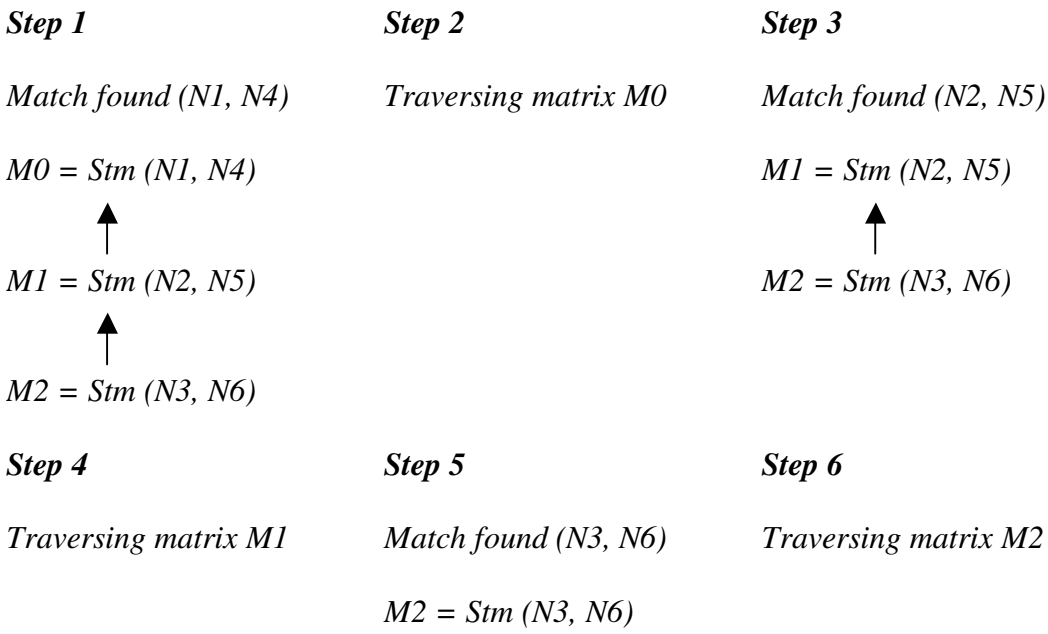


Figure 6. The steps of Simple tree matching

At the end of this example, the algorithm has produced six matrices. This is a waste since it actually only needs three. **Simple tree matching** algorithm uses a dynamic programming scheme to produce these matrices with the aim of finding out the number of nodes in the largest common subtree between two trees. Therefore it does not need to keep any of the matrices once it is done with them. Our algorithm on the other hand does not worry about the number of pairs, but rather identifies the pairs. To do that, we need every matrix. And since all the matrices are thrown away, we have to reproduce them.

Simple tree matching algorithm is not ideal for our purpose, since its purpose is to find the number and not the actually largest common subtree. But we need the matrices because they are the best may to assure the finding of a maximum matching. A better solution is to go back to basics and use **Sequence matching** algorithm instead. The matrices produced here are fully sufficient since we do not need to know the number of nodes in the largest common subtree. And this algorithm does not need lower-level matrices.

The steps for using the **Sequence matching** algorithm with T1 and T2 as describe in Figure 5 are as follow:

Step 1. A match is found at the roots of the trees T1 and T2. Matrix M0 is then produce by passing the subnodes of N1 and N4 to **Sequence matching**.
Step 2. We traverse matrix M0 as we do in Figure 6.

Step 3. A match is found in matrix M0. We have identified N2 and N5. Then, and only then do we produce matrix M1.
Step 4. We traverse matrix M1.
Step 5. We identify N3 and N6 as a match and we produce matrix M2.
Step 6. We traverse matrix M2.

With this strategy we do not produce a matrix until we really need it and none are wasted, because we only produce matrices in one level at a time. This strategy works also well with our **Combining the matching and identifying algorithm**, which will in this case reduce the steps by half because it generate and traverse the matrices at the same time. But we prefer to use **Sequence matching** algorithm because it does not discard the possibility of finding multiple solutions, see section 4.2.5. We incorporated this strategy in Algorithm 7.

4.3.7. Nodes with unordered and ordered subtrees

Sequence matching is based on the assumption that the order of the element is significant, (1, 2) ≠ (2, 1). The order of the subtrees is also significant. But we sometimes need to ignore that constraint to be able to adapt to Java. E.g. method and field declaration orders are not significant. We want to have (1, 2) = (2, 1). Note that every element is unique, no elements can contain identical symbols, and therefore an element can at most match one element in the other sequence. This can be a problem for Java. A statement can occur several times in method bodies.

We have now two ways of matching nodes, where order is significant and where order is not significant. The methods for matching nodes where order is significant are described in previous chapters. And the method for matching nodes when the order is insignificant is just a matter of comparing every element in one sequence with every element in the other sequence.

```
Algorithm: Tree matching roots (T1, T2)

Node flagA numberA subA = T1
Node flagB numberB subB = T2
T = empty tree, will contain the common tree

if flagA and numberA is not identical to flagB and numberB then return T

if flagA then
        T = Node flagA numberA (Tree matching ordered (subA, subB) )
else
        T= Node flagA numberA (Tree matching unordered (subA, subB) )
endif
return T
```

Algorithm 6. Tree matching roots

In the algorithm **Tree matching roots** we give an algorithm that handles both methods. To do this we introduce a flag. The nodes now contain a number and a flag. The flag of a node is to indicate whether the order of the node's subtrees is significant or not. A match between two nodes is defined as two nodes, one from each tree, containing identical symbols. This means that the flags must be identical and the numbers must be the same.

The first algorithm **Tree matching roots** does is to check whether the two roots of its arguments are identical[1]. If they are not, then it returns an empty tree, meaning that there are no matching nodes between the two trees. Note that the notation "*Node flagA numberA subA = T1*" means that the algorithm uses Haskell-style pattern matching for extracting the information of T1' root node, [5]. Symbol "*flagA*" is the variable bound to the value of the flag of T1. The variable "*numberA*" is bound to the number stored in T1. The variable "*subA*" is the subnodes, or list of subtrees, of T1's root.

The algorithm checks the flag to decide if the order of the subtrees is significant or not. It does not matter whether it checks T1 or T2 because we have established that the flags of the roots are identical. We chose to use T1's values. Once the significance is checked then the algorithm constructs a root node using the information from T1's root. If the order is significant then it passes the two lists of subtrees to **Tree matching order** algorithm. Note that the subtrees are passed as sequences containing the roots of the subtrees. And if the order is not important, then the sequences are passed to the algorithm **Tree matching unordered**. Algorithm **Tree matching ordered** and algorithm **Tree matching unordered** are auxiliary algorithms to algorithm **Tree matching roots**. The result of either one is the largest common subtree minus the root.

Algorithm **Tree matching ordered** respects the order of the nodes because it combines the **Sequence matching** algorithm and **Longest common subsequence** algorithm. The latter algorithm must be modified so it can handle the flags. The algorithm constructs the matrix M by calling the **Sequence matching** algorithm. Note that this is the algorithm for comparing sequences and not trees. The reason for choosing sequence-matching algorithms, is because they do not produce unnecessary matrices, see section 4.3.6. The algorithm traverses the matrix looking for matches. The matching pair of nodes is passed on as arguments to the **Tree matching roots** algorithm. When the recursion to lower levels is done, then it continues on traversing the matrix. When the algorithm terminates, it returns the variable subT, the list of common subtrees, to the **Tree matching roots** algorithm, which puts it together with the root to form a proper tree.

Algorithm **Tree matching unordered** on the other hand does not respect the order of the nodes. It also takes two sequences as arguments. It compares every element in sequence A with every element in sequence B. A match is found when A<i> and B<j> is identical. Then the subtrees where A<i> and B<j> are the roots are passed to the **Tree matching roots** algorithm. The result of the lower recursion is stored in the list subT. The operator "+" joins together the list subT with the result of a recursive call, making a new list of subtrees. At the end the list is putted together with the root in the higher recursion to form the largest common subtree between T1 and T2.

[1] It is only necessary for the flags and the number to be identical for the roots to match.

```
Algorithm: Tree matching ordered (A, B)

subT = empty list of subtrees, will possible contain the list of common subtrees
M := Sequence matching(A, B)
m:= |A|
 n:= |B|
C := 0  ,current length of longest common sequence
subTA_i= the subtree at index i of sequence A
subTB_j= the subtree at index j of sequence B

for i := 1 to m do
        for j := 1 to n do
                if M[i, j] > C then
                        C := M[i, j]
                        if subT is empty tree
                                subT := Tree matching roots (subTA_i, subTB_j)
                        else
                                subT := subT + Tree matching roots (subTA_i, subTB_j)
                        endif
                endif
        end
end
return subT
```

Algorithm 7. Tree matching ordered

Algorithm **Tree matching unordered** also finds the list of common subtrees when given two lists of subtrees, A and B, as arguments. It compares ever element in A with every element in B. When a match is found, that means that A<i> = B<j>, then they are passed to **Tree matching roots** algorithm. Or to be more precise, the subtrees where the nodes A<i> and B<j> are the roots are passed to **Tree matching roots** algorithm. The result stored in the variable subT, which is the list of common subtrees when comparing the sequences A and B. When the algorithm terminates, it returns the variable subT. This list is then joined together with the matching parent of A and B, to form the largest common subtree.

```
Algorithm: Tree matching unordered (A, B)

subT = empty list of subtree, will possible contain the list of common subtrees
m:= |A|
 n:= |B|
subTA_i= the subtree at index i of sequence A
subTB_j= the subtree at index j of sequence B


for i := 1 to m do
        for j := 1 to n do
                if A<i> = B<j> then
                        if subT is empty tree
                                subT := Tree matching roots (subTA_i, subTB_j)
                        else
                                subT := subT + Tree matching roots (subTA_i, subTB_j)
                        endif
                endif
        end
end
return subT
```

Algorithm 8. Tree matching unordered

The main difference between **Tree matching ordered** and **Tree matching unordered** is that the second algorithm does not need another algorithm. For the first algorithm, **Sequence matching** forms the matrix, which it traverses.

They do however have in common the fact that they are not optimized. Algorithm **Tree matching ordered** traverses the whole matrix. As we have discussed earlier, this is not necessary. And algorithm **Tree matching unordered** continues to compare a node from sequence A with the rest of the nodes from sequence B after it has found a match. We know for a fact that a node from sequence A can at most match only one node from sequence B.

It is not difficult to correct the algorithms, but these minor imperfections do not affect the outcome of the algorithm and are more an efficiency problem. We choose to not alter these fundamentals algorithm to correct these flaws.

5. Functional comparison algorithms

In this chapter we present functional, real code fragment of the algorithms introduced in chapter 4. There is of course a big difference between pseudo code and actual Haskell implementation. And not to mention that the fundamental algorithms are constructed in an imperative style and Haskell is a functional language. This may cause some conflicts during the transformation, as we see in section 5.1.3.

5.1. Functions for comparing sequences

In this section we describe functions for matching sequences and also functions for finding the longest common subsequences. We give also an improved functions for finding the longest common subsequence.

5.1.1. Sequence matching

In section 4.3.6 we concluded that the basic sequence-matching algorithm serves our purpose best. We found that with **Sequence matching** algorithm, we save time and space because the algorithm does not produce unnecessary matrices. In this section we present a functional version of that algorithm. We will therefore describe this function in greater detail than the other functions. This function, as with its predecessor algorithm, only computes the matrix M.

sequenceMatching:: (Eq t) => [t] -> [t] -> Array (Int, Int) Int

This is the signature of the sequence matching function. The *"[t] -> [t]"* denotes the two arguments of the function. This function takes two lists of the same type. The lists denote the sequences that we wish to compare. With Haskell functions, we do not need to specify the specific types of the sequences. In this case we simply call them for *t*. By this way the function is polymorphic and can apply to any homogenous kinds of sequences. We will therefore no longer use sequences of integers. The *"(Eq t)"* tell us that the elements *t* are subjects to the Haskell Eq class. This ensures the possibility for comparing the elements in the sequences by using the operator "= =". The result of the function is of type "Array (Int, Int) Int", which is Haskell's representation of a matrix, or to be more precise, of an array. But a matrix is basically two-dimensional array.

Following is the implementation of the function **sequenceMatching**. The values of the elements in the matrix are dependent on the values of others. Therefore the matrix is implemented recursively and the computation starts with the first row and column and proceeds from northwest to southeast. The parallel implementation enables us to refer to the array while it is being computed.

```
1    sequenceMatching A B = M
2        where  M = array ( (0, 0), (m, n) )
3                    (       [ ( (i, 0), 0)  |  i <- [1..m] ] ++
4                            [ ( (0, j), 0)  |  j <- [1..n] ] ++
```

```
5                [ ( (i, j), v ) | i <- [1..m], j <- [1..n] ]                )
6        v = returnMax (      ( M!(i-1, j) )
7                             ( M!(j-1, i) )
8                             ( (M!(i-1, j-1)) + compareValue i A j B) )
```

Function 1. sequenceMatching

In line 2, we see that the bounds are set. In line 3 and 4, we set the first row and column to consist only of zeros. In line 5 we see that the index (i, j) is set to the value v. Each of the lines 3 to 5 creates a list of index-value pairs. They are joined together by the infix list concatenation operator "++" to form the whole list of index-value pairs that creates the array. The value v is the result of the function **returnMax**. The signature of **returnMax** is as follow.

returnMax :: Int -> Int -> Int -> Int

Function **returnMax** takes three arguments of type integer and returns the one with the highest value. This function works the same way as Prelude's **max**, which only takes two arguments. The arguments given to **returnMax** are shown in line 6 to 8. The first two arguments are the values of the elements situated immediately to the north and west of the current element. The third argument is the sum of the value of the immediate element in the northwest and the result of the function **compareValue**.

compareValue ::(Eq t)=> Int -> [t] -> Int -> [t] -> Int

Function **compareValue** compares the value at index i in sequence A and the value at index j in sequence B. If the values are identical then it returns 1 otherwise it returns 0. Function **compareValue** uses a built in function to extract the right element from the sequences. And again we see the *"(Eq t)"*. This means that the elements of type *t* inherits from class Eq and function **compareValue** is able to apply the operator "= = " no matter what type the element really have.

In other words, the function **sequenceMatching** creates a matrix of size (m+1)×(n+1), where the first row and column are zeros. The value v of the element (i, j) is always equals to the highest value of its predecessors (in term of a flattened M, see section 4.1.2), until there is a match in A!i and B!j. In that case the value v is incremented by 1. This corresponds to **Sequence matching** algorithm.

5.1.2. Common subsequence

This function sets out to find the longest common subsequence by analyzing the matrix M produced by function **sequenceMatching**. It goes through every element in the matrix and retrieves the information needed to construct the longest common subsequence. The function starts its search at M!(0, 0) and works its way row by row to M!(m, n), where m and n is the length of sequence A and B. The path of the function is best explained by imaging the matrix as a long sequence, a flattened M, see section 4.1.2. The function simply starts at the beginning and works its way to the end.

commonSubsequence :: Array (Int, Int) Int -> (Int, Int) -> Int -> [t] -> [t]

The function **commonSubsequence** returns an array of elements of type *t*. Type t is of course the type of the two sequences A and B, which are given to the function **sequenceMatching** as arguments. The four arguments are a matrix, a pair of bound, an integer and a list of elements. The first argument is the matrix M that **sequenceMatching** returns. The second argument is a pair of bound telling the function **commonSubsequence** how far the function has computed. The third argument is the length of the current longest common subsequence that the function **commonSubsequence** has computed. The last argument is either sequence A or B. It does not matter which one, because it is only use to look up the common elements.

```
1  commonSubsequence M (i, j) C A
2      |    j > n          = commonSubsequence M  (i+1, 0) C A
3      |    i > m          = [ ]
4      |    C < C_         = [ ( A !! (i-1)) ] ++ commonSubsequence M (i, j+1) C_   A
5      |    otherwise      = commonSubsequence M (i, j+1) C A
6           where
7                  C_ = M!(i, j)
```

Function 2. commonSubsequence

The initial invocation of this function is *"commonSubsequence M (0, 0) 0 A"*. The reason the pair of bound is set at (0, 0) is, as mention earlier, that we want the function to start at the beginning of the flattened M. One might wonder why we do not start the traversing with element M[1, 1]. After all, the first common element cannot be found earlier[1]. But even though we do not traverse the first row at all, we still have to traverse the first column of the remaining rows and they are still zeros. We do not improve the efficiency much in the long run. Therefore we leave the efficiency improving to the function **commonSubsequence2** that we present in section 5.1.4.

The variable C is initially zero and as the function computes; variable C increases every time the function finds a match. The guard in line 2 applies if the function is at the end of a row in the matrix. It then calls itself recursively and tells the function to start at the beginning of the next row. The variable C remains the same. The second guard in line 3 applies if the function has finished with the last row and has therefore gone through the whole matrix. It then returns an empty list. The guard in line 4 applies if a match is found. It recognizes a match by comparing the value of the current element (i, j) with variable C. Remember that the variable C denotes the length of longest common subsequence until now. If the value of variable C_, which is computed in line 7, is higher than the value of variable C, then the common element is added to the final list and joined together with other common elements that are to be computed. Note that when the function calls itself recursively this time, the variable C_ , not variable C, denotes the length of the current longest common subsequence. The last guard applies if the other fails, and basically what it does it to call itself recursively telling it to move to the next element in the row.

[1] The first row and column of the matrix are by definition zeros.

We see that in line 4 that we wish to obtain element i in sequence A, because we know that the element is a match. This is done with the built in operator "!!" and note that we decrement the index i. This is because our sequences cognitively start with index 1, as in does in the matrices, but the Haskell lists starts at index 0. We could just as well have taken element j in sequence B, because they are identical. Note that we do not check for case of index out of bounds. If is not possible to be out of bounds because our sequences and the matrix operates with the same boundaries.

5.1.3. Combining the matching and identifying functions

In section 4.2.5 we gave an algorithm for combining the two algorithms **Sequence matching** and **Longest common subsequence**. We cannot give a functional version of that algorithm. Combining the two functions **sequenceMatching** and **commonSubsequence** in their current form is not possible. The algorithm **Combining the matching and identifying algorithm** suggests that during the construction of the matrix, the common elements are retrieved immediately after a match has been identified. Unfortunately this is not possible using the *array* construction in Haskell. There are no side effects in Haskell and we are not able to store the common elements outside the matrix during its computation.

However, it is possible to combine these two functions if we alter the function **sequenceMatching.** If we choose to construct the matrix in a different way such that we are able to retrieve the common elements as we compute. One way of doing this is to first construct a matrix consisting of only zeros. The size of the matrix depends on the size of the sequences to be match, plus an extra row and column. Then we traverse it in the same manner as we do in the function **commonSubsequence**. For each and every element, except the ones in the first row and column, we compute values as usual. And when we find a match, we add it to the longest common subsequence list.

combinedMatchingIdentifying :: [t] -> [t] -> Array (Int, Int) Int ->(Int, Int)-> Int -> [t]

The first two arguments are the two sequences to be compared. The next three arguments correspond exactly to the first three arguments of the function **commonSubsequence**. The function **combinedMatchingIdentifying** must also call itself recursively in the same way as the function **commonSubsequence** does**.** The result of the function is of course the longest common subsequence.

updateMatrix :: Array (Int, Int) Int -> (Int, Int) -> Int -> Array (Int, Int) Int

The function above aids the operation of replacing the existing value of an element. What it does is basically copy every element from the array given as argument, except the element in position of the second argument. It replaces that element's value with its third argument before returning the new matrix. The combined function must do this operation for every element and provide this new matrix as argument every time it calls itself recursively.

The initial invocation for this function is as follows.

combinedMatchingIdentifying A B mZeros (0, 0) 0

Let A and B be two sequences and m = | A | and n = | B |.

mZeros = array ((0, 0), (m, n)) [((i, j), 0) | i <- [1..m], j <- [1..n]]

This solution works, but the implementation of this function is complicated and not compact and easy to understand. It is also space consuming because every time we compute an element and store it in the matrix, we in reality creates a new matrix. In Haskell, we do not have variables in the imperative languages sense, but rather like constant. A variable is bound to a value, and the value does not change once bound [4]. In order to "change" the value, we have to create a new variable with the desire value.

Because of this reason and because it discard the possible of finding multiple solutions as discussed in section 4.2.5, we choose not to implement and use the combined function. We will not show how such a function might be implemented but rather give a sketch. We present this solution because it may prove to be useful in future works.

5.1.4. Improving the commonSubsequence function

In section 4.2.6 we discussed ways to reduce the running time by improving the efficiency. In this section we give a function that not only incorporate the suggestion in section 4.2.6 but we also improve the efficiency in some special cases that are likely to occur, like multiple solutions and broken subsequences.

commonSubsequence2 :: Array (Int, Int) Int -> (Int, Int) -> Int -> [t] -> [t]

The signature of this function is identical to that of the original function. The construction of the function is also similar. We have just added two more guards and done some minor adjustments. The main difference is the order in which the elements are searched. In this function we do not search through every element. We start the search at index M[1, 1] and move on to M[2, 2], M[3, 3] and so on. The extra guards ensure that we do not miss any matches. Let A and B be the sequences to be compared and m = |A|, n=|B|.

```
1   CommonSubsequence2 M (i, j) C A
2     |     j > n          = commonSubsequence2 M  (i+1, 0) C A
3     |     i > m          = [ ]
4     |     C < C_         = [(A !! i)] ++ commonSubsequence2 M (i+1, j+1) C_ A
5     |     C==C_ &&  j /= n  && C < Cj = commonSubsequence2 M (i, j+1) C A
6     |     C==C_ &&  i /= m  && C < Ci = commonSubsequence2 M (i+1, j) C A
7     |     otherwise      = commonSubsequence2 M (i+1, j+1) C A
8           where
9                 C_ = M!(i, j)
10                Cj = M!(i, j+1)
11                Ci = M!(i+1, j)
```

Function 3. commonSubsequence2

Note in line 4 and 7, when we call the function recursively, the pair of indexes is set to (i+1, j+1) and not (i, j+1) as it is in the original function. We do this to exploit the knowledge we have of matrix M, see section 4.1.2. We exploit the fact that if a match is found at index (i, j), then the next match does not occur until at least at index (1+1, j+1). This adjustment saves us from looking at the rest of the row. The next guard, in line 5 ensures that the function always choose the first[1] solution if there are multiple longest common subsequences.

A1 = (1, 2)
B1 = (2, 1)
M1 = sequenceMatching (A1,B1)

A2 = (5, 6, 7)
B2 = (5, 8, 6)
M2 = sequenceMatching (A2, B2)

M1

i \ j	**0**	**1**	**2**
0	0	0	0
1	0	0	1
2	0	1	1

M2

i \ j	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	1	1	1
2	0	1	1	2
3	0	1	1	2

The matrix M1 depicts a situation where there are two possible solutions of the longest common subsequence, M1[1, 2] and M1[2, 1]. The search starts as mention at M1[1, 1]. Without this guard, the function will move on to M[2, 2] and it will consider that element as a match, which it is not. The guard moves the focus to the element M1[1, 2] if it considers the element to be a match. The lazy evaluation style of Haskell and the test "j /= n" ensure that the function stays within bounds when computing the variable *Cj*. Matrix M2 depicts another situation which is also caught by the guard. The guard in line six prevents the wrong element to be identified in a situation described below.

A3 = (1, 4, 2)
B3 = (1, 2, 5)

M3 = sequence matching (A3, B3)

M3

i \ j	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	1	1	1
2	0	1	1	1
3	0	1	2	2

[1] The first solution is the one with the lowest index, see section 4.1.1.

Without the guard, the function would identify the elements M3[1, 1] and M3[3, 3] as the longest common subsequence of A3 and B3, which is wrong. The correct elements in the longest common subsequence between sequences A3 and B3 are M3[1, 1] and M3[3, 2].

5.2. Functions for comparing abstract syntax trees

With sequences we operated with a general type *t*. We are not able to do that when we implement the functions that compare abstract syntax trees. In section 4.3.7 we introduce the data type for nodes in a tree. We will in this section elaborate more of the data representation of trees in our implementations. Here is a Haskell data type for trees.

data Tree = Empty | Node Bool Int [Tree]

A tree is either *Empty*, which means that there are no nodes in that tree, or it consists of a node with a list of subtrees. And further more we see that a node also consists of a flag of type Boolean and an integer. The flag determines how we treat the list of subtrees. If the Boolean value is true, then we consider the order of subtrees to be significant, and insignificant if false. The integer represents the information we want to store in the abstract syntax tree. An empty list means that the node is a leaf.

The function **compareNodes** takes two nodes and returns the value true if they are equal, otherwise it returns false. Two nodes are considered equal if the pair of Boolean-values, one of each node, is identical and the pair of integers are identical. The lists of subtrees are not considered when comparing nodes.

compareNodes : : Node -> Node -> Bool

5.2.1. Tree matching

This functions task is to compare the roots of two trees. It is a functional version of algorithm **Tree matching roots.** It works in the same way; first it checks whether the roots are identical and then sends the subtrees to the appropriate functions for comparison; **orderedTreeMatching** or **unorderedTreeMatching**.

treeMatchingRoots :: Tree -> Tree -> Tree

The function **treeMatchingRoots** takes two trees as arguments and returns the largest common subtree. If at least one of the trees is *empty* then it returns an *empty* tree. It also returns *empty* if the roots do not match.

```
1   treeMatchingRoots  Empty  _ = Empty
2   treeMatchingRoots  _  Empty = Empty
3   treeMatchingRoots a@(Node flgA intA treesA) b@(Node flgB intB treesB)
4       | cmpN && flgA = Node flgA intA (treeMatchingOrdered treesA treesB)
5       | cmpN && not flgA = Node flgA intA (treeMatchingOrdered treesA treesB)
```

```
6           | otherwise = Empty
7           where cmpN = compareNodes a b
```

Function 4. treeMatchingRoots

The guards in line 1 and line 2 return an empty tree if one of the fundtion's arguments is empty. The operator "@" in line 3 assigns the pattern after it to the symbol before it. We can then use the symbol when we address the whole pattern. In line 4 the variable *cmpN* is the result of a matching between the nodes a and b, which is computed in line 7. If the result is the value true and the flag of node a (and of node b) is also true, then the function constructs a root node and sends the subtrees of a and b to **orderedTreeMatching**. If the flag is false then a root node is then also constructed, but the subtrees are sent to **unorderedTreeMatching**.

5.2.2. Ordered tree matching

In section 4.3.6 we suggested a strategy where we do not need to create unnecessary matrices and in section 4.3.7 we incorporated that strategy in algorithm **Tree matching ordered**. In this section we present a function that respects the order of the element in a sequence and also incorporate the mentioned strategy.

orderedTreeMatching :: [Tree] -> [Tree] -> [Tree]

This function takes two lists of subtrees and returns a list of common subtrees. Note that this function does not handle the nodes that are the parents of these subtrees because they are compared at a higher level of recursion. We can say that this function corresponds to the algorithm **Tree matching ordered**, because it also needs a sequence-matching algorithm. The function **orderedTreeMatching** calls the function **sequenceMatching** to compute. The function **sequenceMatching** uses the function **compareValue** and a few minor adjustments are needed, since we do not deal with sequences of integers any more. Earlier we compared elements of type *t*'s by using the operator "= =". This will not have the desire effect with nodes because we do not wish to include the subtrees in the comparisons. Therefore the function **compareValue** must call the function **compareNodes** to determine whether the nodes are equal or not.

```
1   orderedTreeMatching  [ ]  _ =  [ ]
2   orderedTreeMatching  _  [ ] =  [ ]
3   orderedTreeMatching treesA treesB = orderedCommonSubTrees M (1, 1) 0 treesA treesB
4           where  M = sequenceMatching treesA treesB
```

Function 5. orderedTreeMatching

The first two lines returns an empty list if either of the two arguments are also empty. We see the function generates a matrix M in line 4 by calling **sequenceMatching.** Matrix M is then given to the function **orderedCommonSubTrees**, which corresponds to the function **commonSubsequence2.** Except that the last function do not compute into lower levels of recursion when it finds a match.

orderedCommonSubTrees :: :: Array (Int, Int) Int -> (Int, Int) -> Int -> [t] -> [t] -> [t]

We see that the list of arguments is almost exactly the same function **commonSubsequence2**. An additional list has been added as argument. The two last arguments are the two sequences used to generate the matrix M. They are needed for looking up elements when a match is found. The implementation is also similar to the function **commonSubsequence2.**

```
1   orderedCommonSubTrees M (i, j) C A B
2     |   j > n      = orderedCommonSubTrees M  (i+1, 0) C A B
3     |   i > m      = [ ]
4     |   C < C_    = [Node_ ] ++ orderedCommonSubTrees M (i+1, j+1) C_ A B
5     |   C==C_ &&  j /= n  && C < Cj  = orderedCommonSubTrees M (i, j+1) C A B
6     |   C==C_ &&  i /= m  && C < Ci = orderedCommonSubTrees M (i+1, j) C A B
7     |   otherwise       = orderedCommonSubTrees M (i+1, j+1) C AB
8         where
9               C_ = M!(i, j)
10              Cj = M!(i, j+1)
11              Ci = M!(i+1, j)
12              Node_ = treeMatchingRoots T1 T2
13              T1 A !! i
14              T2 = B !! j
```

Function 6. orderedCommonSubTrees

The main difference in this version is in line 4. When a match is found we do not just return the common element, but we construct a new node. In line 12 we construct a node using the values of the common node in sequence A. The two common nodes are then sent to function **treeMatchingRoots** as arguments. They are the roots of a new common subtree. In line 13 and 14 we get the nodes from the lists and extract the information we need to form the new node in line 12.

When this function terminates, it has formed a list of trees. This list is then put together with the root node to form the largest common subtree.

5.2.3. Unordered tree matching

This function matches lists of trees where the order of the trees is not significant. This is a step towards to Java. An example of such a list is the list of methods in a class. In Java, the order of the methods is irrelevant. Therefore we cannot use the functions that incorporate Yang's matrices, such as functions **orderedTreeMatching.**

In section 4.3.7 we introduce an algorithm, Algorithm 8, where we match two lists against each other without considering order. The function in this section is a functional version of that algorithm.

unorderedTreeMatching :: [Tree] -> [Tree] -> [Tree]

As with function **orderedTreeMatching**, this function takes two lists of trees and returns the list of common trees. And this function does not handle the roots that these subtrees belongs to because they are compared at a higher level of recursion, in the function **treeMatchingRoots**.

```
1    unorderedTreeMatching treesA treesB
2        |    treeA = = [ ]  | |  treesB = = [ ]    =    [ ]
3        |    otherwise      =   commonSubseqUnordered treesA treesB treesB
```

Function 7. unorderedTreeMatching

The first guard in line 2 returns an empty list if one of the arguments is also an empty list. The function then calls the auxiliary function **unorderedCommonSubTrees**. Note that the two last arguments of that function are the same. The reason that this function does not do much is that we want to encapsulate the complicated implementation in an auxiliary function and provide an intuitive interface; given two lists we get one list of common elements.

unorderedCommonSubTrees :: [Tree] -> [Tree] -> [Tree] -> [Tree]

This function takes three lists and as mentioned, the last two are initially identical. The reason for this is that the first two are used for comparison during recursive calls. During these calls the second list undergoes various manipulations, and sometimes we need to go back to its original value. Therefore the last argument is not subject to any change at all.

```
1  unorderedCommonSubTrees [] _ _  = []
2  unorderedCommonSubTrees (_:tlA) [] orgB = unorderedCommonSubTrees tlA orgB orgB
3  unorderedCommonSubTrees a@( hdA : tlA )  ( hdB: tlB )  orgB
4      | compareNodes hdA hdB = [ Node_ ]++ unorderedCommonSubTrees tlA orgB orgB
5      | otherwise   = unorderedCommonSubTrees a tlB orgB
6            where Node_ =  treeMatchingRoots hdA hdB
```

Function 8. unorderedCommonSubTrees

The first line in Function 8 applies when the function has finished computing. The strategy of this function is to compare the first element in the first list with every element in the second list. When it has gone through the second list, it moves on to the second element in the first list and compares it with the entire second list and so on. Therefore when the first list is empty, it means that we have compared every element in it and the function terminates. Note that when the function calls itself and gives a tail of a list, then the head is lost.

The second line applies when the function has gone through comparing every element in the second list with the first element in the first list. It then calls itself recursively with the rests of the first list (*tlA*), and two instances of original second list *orgB* as arguments. The reason for using the list *orgB* as the second arguments is give the rest of the first list a chance to compare with the entire second list.

The guard in line 4 applies if the result of the function **comparingNodes** is true. This means that there is a match between the two heads of the two lists and the variable *Node_* is

constructed in line 6. Variable *Node_* is the root of the largest common subtree when comparing the two heads of the lists. This subtree is then joined together with the rest of the possible subtrees.

As with the function **unorderedTreeMatching,** the result of this function is put together with the root node in the higher recursion to form the largest common subtree.

5.3. Functions for maximum matching

We present in this section two functions that are in a sense the API for the functions described in the previous sections. These two functions both take two lists as arguments and return a third. The implementation of our tool does not call the functions in the previous sections directly, but only these two we present here.

5.3.1. Ordered maximum matching

This function combines the two functions **sequenceMatching** and **commonSubsequence2.** This combined function is then a useful auxiliary function as we see in section 6.2.

```
1.  orderedMaximumMatching :: (Eq t) => [ t ] -> [ t ] -> [ t ]
2.  orderedMaximumMatching A B = commonSubsequence2 M (1, 1) 0 A
3.          where M = sequenceMatching A B
```

Function 9. orderedMaximumMatching

This function takes two lists of the same type and returns a third, which denotes the longest common subsequence when comparing the two arguments. The order of the elements in the lists is significant. The elements in the list are subjects of the standard Haskell class *Eq* and we are able to apply them to the equality oprator "= = " when comparing them.

5.3.2. Unordered maximum matching

This function does not respect the order of the elements of the lists when comparing them. It compares every element from the first list with every element from the second list. This function is a variation of the function **unorderedCommonSubTrees.**

```
1.  unorderedMaximumMatching :: (Eq t) => [ t ] -> [ t ] -> [ t ]
2.  unorderedMaximumMatching A B = auxUMM A B B
3.  auxUMM :: (Eq t) => [ t ] -> [ t ] -> [ t ] -> [ t ]
4.  auxUMM [ ] _ _ = [ ]
5.  auxUMM ( _ : tlA ) [ ] orgB = auxUMM tlA orgB orgB
6.  auxUMM a@(hdA : tlA ) ( hdB : tlB ) orgB
```

```
7.      | hdA = = hdB       = [ hdA ] ++ auxUMM tlA orgB orgB
8.      | otherwise        = auxUMM a tlB orgB
```

Function 10. unorderedMaximumMatching

The **unorderedMaximumMatching** is in reality two functions; the function **unorderedMaximumMatching** and the auxiliary function **auxUMM**, which is short for auxiliary unordered maximum matching. Function **unorderedMaximumMatching** provides a simple API and hides the computation details of **auxUMM**. The first function corresponds to the function **unorderedTreeMatching** and the second function corresponds to the function **unorderedCommonSubTrees**, in the sense that they both treat subnodes where order is not significant and that the function **unorderedTreeMatching** needs the function **unorderedCommonSubTrees** to compute in the same way as the function **unorderedMaximumMatching** need the function **auxUMM**.

In line 2 we see that the only function **unorderedMaximumMatching** does is to call upon auxiliary function **auxUMM**, which does all the comparison work. The construction of auxiliary function **importDeclarationList** is exactly the same as the function **unorderedCommonSubTrees.** The only difference is in line 7. In this case we do not need to construct a common subtree but we only returns the first argument if the two arguments matches. This function is only use to compare lists where order is insignificant and where we do not need to compare subterms of the elements of the lists.

5.4. Comments

During the construction of these functions, we are not concerned much with the efficiency of the functions. The most important for us is that the functions compute correct values and that the functions are neat and compact. In section 3.3 conclude that this tool is far from ready for commercial use, but we do not totally neglect the efficiency. In section 4.2.6 and 5.1.4 we implement functions that are more efficient than the basic functions. But these changes are not done with the intention of reducing the running time, but because it is a waste of time and space if we do not. The changes are an act of principle and it is a bonus that it also reduces running time.

The reason for choosing Haskell as the programming language shines through during the construction of these functions. The pattern-matching feature of Haskell is powerful yet easy to use. The functions owe their compact and nice structures much to pattern matching.

6. Implementation

6.1. Overview

The work of finding the syntactic difference between two Java programs is not just the task of finding the largest common tree between two abstract syntax trees, although it is the hardest and most time consuming. In this section we give a description of the different stages in the procedure of comparing two programs.

The steps are as follow.

1) Preparing source files.
2) Constructing abstract syntax trees.
3) Identify methods inherited from interfaces.
4) Comparing abstract syntax trees; finding the largest common subtree.
5) Analyzing the common sub tree.
6) Generating a report on the differences.

The preparation of source files makes Strafunski able to generate abstract syntax trees. The most important is that the source files are syntactically correct, which we assume that they are. It is also important for all the associated files, like source files for super classes or interfaces, to be accessible to Strafunski. We need the abstract syntax trees for these files to identify contextual differences.

Strafunski then takes in the second stage. It generates not only abstract syntax trees for the files that we want to compare, but also for all associated files. Our task here is to separate these trees and getting the ones we wants. Then we traverse the two trees and search for methods that they inherited from their implemented interfaces.

When we have the two abstract syntax trees prepared as we want, the job of comparing these can begin. This stage is described throughout this thesis since it the most important stage and requires complicated algorithms.

The largest common subtree is not the final result. To obtain the final result, we need to compare the common subtree with the original abstract syntax trees. Let T1 and T2 be version 1 and version 2 of the program we want to compare. And T' be the largest common subtree when comparing T1 and T2. Then we get the desired information if we compare T1 against T' and T2 against T'. The information that we are looking for is the "rest" when "subtracting" T' from T1 and from T2. In other words, we are looking for the nodes in either T1 or T2 not found in T'. The result of the comparison between T' and T1 denotes deletions in going from T1 to T2. And the result from the comparison between T' and T2 denotes the additions in going from T1 to T2. Some changes might be considered as an alteration and neither an addition nor a subtraction, but an alteration is a subtraction-additional pair of actions. If a node in T' has a corresponding node in both T1 and T2, then the node in T1 has changed into the node in T2. Note that we said corresponding, not matching. This is best explained by an example.

Let node N1 in tree T1 have the value *integer* and node N2 in tree T2 have the value *double*. Node N1 does not match N2, so the largest common subtree T' should not have a node that matches any of the nodes N1 or N2. But if we want to match N1' subnodes to N2's subnodes anyway, we need to go away from the rule that the parent nodes must match. For this reason there must be a node in the corresponding place in T' as N1 has in T1 and N2 has in T2, then we must assign the node N' in T' with a value that reflect the situation. We use a third value to achieve this. Let N' have the value *changed* in this example. Then when comparing N' in T' with N1 in T1, we detect that N1 has been deleted since it does not match with the corresponding node in T'. And when comparing N' in T' with N2 in T2, we detect that N2 has been added because it does not match its corresponding node N' in T'. We view the deletion action and addition action as a single alteration action, since the two actions derive from the same node in T'

The final stage is to generate a reader-friendly report that sums up the changes between the two versions of the program. This thesis does not include this stage. The outputs of our functions contain all the information needed to produce the best report of differences, but it is in the form of an abstract syntax tree. It is a trivial exercise to transform the outputs into a reader-friendly report.

6.2. Comparing Java abstract syntax trees

In this section we present and describe the function that does the actual work of comparing two Java programs. We do not show all of the implementation because it takes a lot of time and space, and we will not gain much by doing that. What this section does is to explain the most important parts of the implementation and we go through a few examples that are representative for the rest of the implementation.

In section 4.3.7 we introduce a flag to distinguish between ordered and unordered subtrees. That flag demonstrates how the functions interact with each other. When comparing abstract syntax trees generated by Strafunski, we do not have such a flag. The Java abstract syntax tree is also complicated. Comparing the Java abstract syntax trees is not as simple as sending them as arguments to the function **treeMatchingRoots** and let the function do the rest. The abstract syntax trees are not structured the same way as our simple and basic trees are. It will require more work to be able to compare Java programs. We have to adapt the functions to Java syntax.

This does not mean that the effort of previous chapters is wasted. The fundamental algorithms that lead to the corresponding functions are still the foundation on which our tool rests. What this means is that, since there are no flag to decide whether the order is significant or not, we have to make those decisions in a case-to-case basis. And the functions explained in this section demonstrate the different cases and how we solve the problems.

Strafunski breaks down a Java program into small and categorized data types, or data types. These data types vary in size and complexity therefore we cannot create a universal function that traverses through the abstract syntax tree and compares all the different kind of data types or nodes. We have to create a function for each and every kind of data type. We have to analyze what the data types represent and decide whether the order of the subtrees is significant or not, and then implement the function that compares them.

The consequence of this is that we now have three sorts of functions; order is significant, order is insignificant and data type specific. The two first sorts have been described in details in the previous chapters. The data type specific functions cannot be described in general terms, but explained with examples. The following sections demonstrate our solution on key areas.

6.2.1. Comparing compilation units

A compilation unit represents basically a Java file. Most times such a file consists of the implementation of one class, but one may find implementation of several classes and interfaces. The compilation unit is to the root node of an abstract syntax tree. Strafunski represents compilation units as follows.

data *CompilationUnit* = ***CU*** *(Maybe PackageDeclaration) [ImportDeclaration] [TypeDeclaration]*

The characters *CU* is the constructor for the data type *CompilationUnit*. As we see, a compilation unit consists of an optional package declaration, a list of import statements and a list of type declarations. The data type *TypeDeclaration* is either a class declaration, an interface declaration or an empty block statement (semicolon). The function **compilationUnit** compares the two roots of the abstract syntax trees.

```
1  compilationUnit :: CompilationUnit ->CompilationUnit ->CompilationUnit
2  compilationUnit (CU pdA impsA tpsA) (CU pdB impsB tpsB)
3  = CU (packageDeclaration pdA pdB) (importDeclaration impsA impsB)
         (typeDeclarationt tpsA tpsB)
```

Function 11. compilationUnit

The data type *CompilationUnit* has not just one set of subtrees, but also three. The function for comparing compilation units is simple because the data type does not have any data for comparison. The function just returns a new compilation unit. In line 3, we see that the constructor *CU* and function invocation for three function; **packageDeclaration**, **importDeclaration** and **typeDeclaration**. These functions' task is to compare the sub data types they are specifically designed for.

We break down the compilation units in line 2 by applying pattern matching. The variables are then sent as arguments to the three functions used to construct a new compilation unit. The two package-declarations *pdA* and *pdB* are sent to the function **packageDeclaration**, the two lists of import statements are sent to the function **importDeclaration** and finally the two lists of type declarations are sent to the function **typeDeclaration**. The results of these functions form the new compilation unit.

We see in the compilation unit construction that there is only one instance of package declaration. The function for comparing package declarations cannot therefore be a subject of

list comparison. The function **packageDeclaration** is therefore a data type specific function. Strafunski implements package declarations as follows.

```
data PackageDeclaration = Package_semicolon Name
                          deriving (Eq)
```

The line "*deriving (Eq)*" is our supplement and means that the data type *PackageDeclaration* is a subject of the class Haskell standard class Eq and that we can apply *PackageDeclaration* to the equality operator "= =". The result is that the implementation of the function **packageDeclaration** is compact.

```
1   packageDeclaration :: Maybe PackageDeclaration ->Maybe PackageDeclaration
                       ->Maybe PackageDeclaration
2   packageDeclaration pkdA pkdB
3                      | pkdA  == pkdB      = pkdA
4                      | otherwise          = Nothing
```

Function 12. packageDeclaration

The function takes on two package declarations as arguments and returns its first argument them, in line 3, if they are identical. Otherwise it returns *Nothing*. In Java, one is not force to declare packages, therefore the Maybe constructor in the signature. The Maybe constructor is defined in Haskell as:

```
Maybe a = Nothing | Just a
```

The lists of import statements are subjects of insignificant order matching because the sequence of import statements in Java is not important. Strafunski implements import declaration as follows.

```
data ImportDeclaration = SingleTypeImportDeclaration SingleTypeImportDeclaration
                       | TypeImportOnDemandDeclaration TypeImportOnDemandDeclaration
```

The import declaration list contains data types of the types *SingleTypeImportDeclaration* or *TypeImportOnDemandDeclaration*.

The function for comparing import declarations is as follows.

```
1   importDeclaration :: [ ImportDeclarationList ] -> [ ImportDeclarationList ] ->
                         [ImportDeclarationList]
2   importDeclaration a b = unorderedMaximumMatching a b b
```

Function 13. importDeclaration

All the function **importDeclaration** does is to call on the function **unorderedMaximumMatching,** which does all the work. Comparing further down into lower levels of the data type *ImportDeclaration* does not give us any more accurate and

desirable report of differences. Therefore we are able to use the function **unorderedMaximumMatching**.

The lists of the data type *TypeDeclarations* are also subject to order insignificant matching. But we cannot pass on the arguments to the function **unorderedMaximumMatching** as the function **importDeclaration** does. We need to compare subtypes of *TypeDeclaration* and therefore we have to specific implement the function for comparing *TypeDeclaration*. Strafunski implements type declaration as follows.

```
data TypeDeclaration = ClassTypeDeclaration ClassDeclaration
                     | InterfaceDeclaration InterfaceDeclaration
                     | Semicolon5
```

Since the data type TypeDeclaration consists of either three subtypes, the implementation of the function **typeDeclaration** is also divided threefold.

```
1    typeDeclaration :: [TypeDeclaration] -> [TypeDeclaration] -> [TypeDeclaration]
2    typeDeclaration a b = typeDeclarationList a b b
```

The function **typeDeclaration** provide a simple API and corresponds to the function **unorderedTreeMatching.** Therefore in line 2 it just passes the argument on to the auxiliary function **typeDeclarationList**.

```
3    typeDeclarationList :: [TypeDeclaration] -> [TypeDeclaration] ->
                 [TypeDeclaration] -> [TypeDeclaration]
4    typeDeclarationList [ ] _ _ = [ ]
5    typeDeclarationList (hd : tl) [ ] orgB = typeDeclarationList tl orgB orgB
```

These lines apply to the condition that terminates the function. In the next segment of the function we matches the first subtype of type declaration, class declaration. This segment of the function only applies if the heads of both lists are of type *ClassTypeDeclaration*.

```
6    typeDeclarationList  a@((ClassTypeDeclaration cdA) : tailA)
                 ((ClassTypeDeclaration cdB) : tailB)  orgB
7      | idA = = idB  =  [ ClassTypeDeclaration (classDeclaration cdA cdB) ] ++
                 typeDeclarationList tailA orgB orgB
8      | otherwise    = typeDeclarationList a tailB orgB
9            where
10                 idA = getFirstId cdA
11                 idB = getFirstId cdB
```

The construction of this segment is exactly like the function **unorderedCommonSubTrees**. The two calls for function **getFirstId** in line 10 and 11 returns the name of the two classes. These functions are addressed in section 6.3. The two classes are considered to match if and only if the names are identical. The call of function **classDeclaration** in line 7 returns a complete match for the subentities of data type *ClassTypeDeclaration*. The function **classDeclaration** will not be described in this thesis.

The next segment applies the second subtype of the data type *TypeDeclaration*; interface declarations. Again the next segment only applies if both the head of the two lists are both of the same type. This segment of the function applies if types are *InterfaceDeclaration*.

```
12    typeDeclarationList  a@((InterfaceDeclaration ifdA) : tailA)
                  ((InterfaceDeclaration ifdB) : tailB)  orgB
13    | idA = = idB  =  [ InterfaceDeclaration (interfaceDeclaration cdA cdB) ] ++
                  typeDeclarationList tailA orgB orgB
14       | otherwise    = typeDeclarationList a tailB orgB
15                where
16                      idA = getFirstId ifdA
17                      idB = getFirstId ifdB
```

This segment works the same way as the previous. The difference is the call of function **interfaceDeclaration** in line 13. That function returns the common data type of type *InterfaceDeclaration* when it matches the two *InterfaceDeclaration* data types. We do not address this function in this thesis.

The last segments rounds it up by matching the last subtype, Semicolon5. This type consists only of the constructor and nothing else. The last two lines applies in the case that the head of the two lists do not have the same type.

```
18    typeDeclarationList  (Semicolon5:tailA) (Semicolon5:tailB) orgB
19    typeDeclarationList  a ( _ :tailB) orgB
20                = typeDeclarationList  a tailB orgB
```

Function 14. typeDeclaration

This concludes this example of matching compilation units. We choose not to follow the trail any further by describing the two functions **classDeclaration** and **interfaceDeclaration**. We gain nothing more by pursuing them. Instead we will give other examples that are subjects to order significant matching.

6.2.2. Comparing method declarations

We show in this section our solution for comparing methods; their signatures and method bodies. According to Sestoft [3], a method declaration declaring method m has the form:

method-modifiers returntype m(*formal-list*) *throws-clause*
method-body

In parallel, the data structure of methods in Strafunski is as follows.

data *MethodDeclaration* = ***MethodHeader_MethodBody*** *MethodHeader MethodBody*

data *MethodHeader* = ***Modifier_s_Type_MethodDeclarator_Throws_opt*** *[Modifier] Type MethodDeclarator (Maybe Throws)*

```
    | Void [Modifier] MethodDeclarator (Maybe Throws)

data MethodDeclarator = Comma2 Identifier [FormalParameter]
                      | MethodDeclarator MethodDeclarator

data MethodBody =   Block1 Block
                  | Semicolon3
                  | Nothing31

data Block = BlockStatement_s  [BlockStatement]
```

As we see, Strafunski breaks down the method declaration into smaller data types. These data types require a function of their own with implementation specific for them. We start with the function for comparing the data type *MethodDeclaration*.

```
1   methodDeclaration :: MethodDeclaration -> MethodDeclaration ->
                MethodDeclaration
2   methodDeclaration (MethodHeader_MethodBody mhA mbA)
                (MethodHeader_MethodBody mhB mbB)
3   = MethodHeader_MethodBody (methodHeader mhA mhB)(methodBody mbA mbB)
```

Function 15. methodDeclaration

This function's only task is to forward the method header parameters to the function **methodHeader** and the method body parameters to the function **methodBody**. In line 3, it creates a new *MethodDeclaration* data type and calls the two auxiliary functions.

The method header has two forms. The two options are similar except that the first option represents methods with return types and the second option represents the ones that do not. Note that the signatures of the two methods being compared have been established to be identical in higher recursion; therefore we know that the methods are the same.

```
1   methodHeader :: MethodHeader ->MethodHeader ->MethodHeader
2   methodHeader (Modifier_s_Type_MethodDeclarator_Throws_opt modsA tpA mdA
        thA) (Modifier_s_Type_MethodDeclarator_Throws_opt modsB tpB mdB thB)
3       = Modifier_s_Type_MethodDeclarator_Throws_opt (modifier modsA modsB)
        (typeMatching tpA tpB) (methodDeclarator mdA mdB) (throws thA thB)
```

The above segment of the function deals only with the first type of methods, the ones that has a return type. The segment only applies if both of the arguments are of this type.

```
4   methodHeader (Void modsA mdA thA) (Void modsB mdB thB)
5       =Void (modifier modsA modsB) (methodDeclarator mdA mdB) (throws thA thB)
```

The above segment is similar to the first segment, but it deals with methods that do not return any values.

```
6    methodHeader (Void modsA mdA thA)
         (Modifier_s_Type_MethodDeclarator_Throws_opt modsB tpB mdB thB)
7        = Void (modifier modsA modsB) (methodDeclarator mdA mdB) (throws thA
     thB)
8    methodHeader (Modifier_s_Type_MethodDeclarator_Throws_opt modsA tpA mdA
     thA) (Void modsB mdB thB)
9        = Modifier_s_Type_MethodDeclarator_Throws_opt (modifier modsA modsB)
     tpA (methodDeclarator mdA mdB) (throws thA thB)
```

Function 16. methodHeader

The last segment of function **methodHeader** applies when one of the methods has a return type and the other does not. Line 6 and line 7 appliers to the combination void-returntype and line 8 and line 9 vice versa. In both cases the function returns a *methodHeader*, which has the form of its first argument. The subentities of the data type *methodHeader* are compared the same way in both cases. Note that in line 9 we return the variable *tpA* without comparing it to anything. Variable *tpA* denotes the return type of a method. Since the second argument in this case does not have a return type, then we use the variable of the first argument. We need a return type in the construction of the data type *MethodHeader* when it is the header of a method that returns a value. This problem is addressed in section 6.1. During the analysis stage, we keep this in mind and therefore able to report that the method in question has changed from returning type *tpA* to not returning anything (void method) from one version to the other.

The next function compares the data type MethodDeclarator. This function is straightforward because the data type contains the signature information of methods. And as mention earlier, the signatures of the methods have been compared in higher recursions and have been found to be identical before they could reach these functions.

```
1    methodDeclarator :: MethodDeclarator ->MethodDeclarator ->
     MethodDeclarator
2    methodDeclarator (Comma2 idA fpsA) (Comma2 idB fpsB)
3        = Comma2 idA (formalParameter fpsA fpsB)
4    methodDeclarator (MethodDeclarator mdA) (MethodDeclarator mdB)
5        = MethodDeclarator (methodDeclarator mdA mdB)
6    methodDeclarator (MethodDeclarator mdA) b@(Comma2 idB fpsB)
7        = MethodDeclarator (methodDeclarator mdA b)
8    methodDeclarartor a@(Comma2 idA fpsA) (MethodDeclarator mdB)
9        = methodDeclaration a mdB
```

Function 17. methodDeclarator

The *MethodDeclarator* data type has two forms and the last one is recursively constructed. The lines 4 to 9 recursively calls on the function until both the *MethodDeclarator* data type has the first form. Lines 2 and line 3 applies when that happens. Since the signatures of the methods have already been compared, there is nothing else the function can do but to return the one of it arguments. Note that the formal parameters are sent to the function **formalParameter** in line 3. The reason is although the types are identical their names might have been changed form one version to the other.

The *MethodBody* data type also has two forms. The first one contains a list of statements and the second is a semicolon, representing the empty body.

```
1    methodBody :: MethodBody ->MethodBody ->MethodBody
2    methodBody (Block1 blckA) (Block1 blckB) =Block1 ( block blckA blckB)
3    methodBody (Semicolon3) (Semicolon3) = Semicolon3
4    methodBody _ _ = Nothing31
```

Function 18. methodBody

The last line is in the case where the above combinations fail. The constructor *Nothing31* is our supplement to Strafunski's implementation. It is used when there is no other alternative to express that the two method bodies in question have nothing in common and an absent of the data type *MethodBody* in the data type *MethodDeclaration* is not incorporated according to the grammar. This kind of problem is addressed in section 6.1.

The last function in this section handles the comparison of statements in a method body. The order of statement is important for the semantic of programs, and the function is therefore subject to the order significant matching.

```
1    block :: Block -> Block -> Block
2    block (BlockStatement_s bsA) (BlockStatement_s bsB)
3                = BlockStatement_s (orderedMaximumMatching bsA bsB)
```

Function 19. block

The implementation of this function is simple. The two lists of statements are sent to the function **orderedMaximumMatching** and let it do all the works.

6.2.3. Comparing class bodies

In the previous sections we have seen example of order significant matching, order insignificant matching and specific matching. This example illustrates more of our use of the Strafunski traverse schemes.

According to Sestoft [3], the classbody may contain declarations of fields, constructors, methods, nested classes, nested interfaces, and initializer blocks. The declarations may appear in any order.

```
{
    field-declarations
    constructor-declarations
    method-declarations
```

```
        class-declarations
        interface-declarations
        initializer-blocks
    }
```

Here are the corresponding Strafunski representations.

```
data ClassBody = ClassBodyDeclaration_s [ ClassBodyDeclaration ]

data ClassBodyDeclaration = ClassMemberDeclaration ClassMemberDeclaration
                          | StaticInitializer StaticInitializer
                          | ConstructorDeclaration ConstructorDeclaration
                          | EmptyClassBodyDeclaration
                          | ClassDeclaration1 ClassDeclaration
                          | ClassDeclaration2 InterfaceDeclaration
```

The data type *ClassBodyDeclaration* has six forms and creating a function that handles all the combination in the same way as the previous functions do, is not an option. The function will be very complicated and cluttered. We present another way, which uses Strafunski traversal scheme extensively.

```
1   classBody :: ClassBody -> ClassBody -> ClassBody
2   classBody (ClassBodyDeclaration_s clsbdsA) (ClassBodyDeclaration_s clsbdsB)
3   = ClassBodyDeclaration_s (clms++st++cs++es++cl1++cl2)
4       where
5           clms = classMemberDeclaration clmsA clmsB
6           st = staticInitializer stA stB
7           cs = constructorDeclaration csA csB
8           es = emptyClassBodyDeclaration esA esB
9           cl1 = classDeclaration cl1A cl1B
10          cl2 = interfaceDeclaration cl2A cl2B
11          [clmsA] = getClassMemberDeclarations clsbdsA
12          [clmsB] = getClassMemberDeclarations clsbdsB
13          [stA] = getStaticInitializers clsbdsA
14          [stB] = getStaticInitializers clsbdsB
15          [csA] = getConstructorDeclarations clsbdsA
16          [csB] = getConstructorDeclarations clsbdsB
17          [esA] = getEmptyClassBodyDeclarations clsbdsA
18          [esB] = getEmptyClassBodyDeclarations clsbdsB
19          [cl1A] = getClassDeclarations clsbdsA
20          [cl1B] = getClassDeclarations clsbdsB
21          [cl2A] = getInterfaceDeclarations clsbdsA
22          [cl2B] = getInterfaceDeclarations clsbdsA
```

Function 20. Class body

This may look overwhelming at first, but is quit easy to get an overview. The lines from 11 to 22 use Strafunski to traverse the classbodies and extract the different subtypes of the data type ClassBodyDeclarations into respectively lists. The functions used to do these operations are

described in section 6.3. For each type of class declarations there are two lists, one of each class body. These lists are then given as arguments to their respective functions for comparison. This is done in line 5 to 10. These functions use various forms of matching, suited for their type, to compare the lists. The results of these operations are then joined together in line 3 to form the finished list of common class body declarations.

We do not explain further the functions in the lines 5 to 10. They all have the same form as the other functions explained in section 6.2.

6.3. Strafunski traversal

Strafunski's main purpose is traversal of abstract syntax trees. Traversal is done by generic functions that can traverse into terms and subterms while mixing uniform and type-specific behavior. Following is a Haskell code skeleton for traversal functions [15]. We give a small description, but a comprehensive explanation can be found at our references, [15] [9] [6].

```
traversal term = apply (scheme step) term
        where step = default 'adhoc' mono1 … 'adhoc' mono2
```

The *apply* is either *applyTP* or *applyTU*, which are explicit application combinators. The *scheme* is a placeholder for traversal schemes, such as *full_tdTU* (type-unified full traversal), *once_tdTU* (type-unified single-hit traversal) and *stop_tdTP* (type-preserved cut-off traversal). We use the *full_tdTU* scheme the most in our function. The *default* is a placeholder for the default type. The common type are: idTP (type preserve), constTU u (uniform type u) and failTU (always failing type). The 'adhoc' is either 'adhodTU' or 'adhocTP'. And $mono_1$ …$mono_2$ are type-specific rewrite steps.

The most used function that incorporates the Strafunski traversal is the function **getFirstId**.

```
1    getFirstId :: (Term t) => t -> String
2    getFirstId cus
3      | id == [] = ""
4      | otherwise = head id
5            where
6            [id] = getIdentifier cus
```

Function 21. getFirstId

This method takes an abstract syntax tree as argument and returns the first identifier that it finds. The actual traversal is done by the function **getIdentifiers**.

```
1    getIdentifier ::(Term t, Monad m) => t -> m [String]
2    getIdentifier cus = applyTU (full_tdTU id) cus
3            where
4            id = constTU [] `adhocTU` idn
5            idn (identifier::Identifier) = return [show identifier]
```

Function 22. getIdentifiers

We see in line 2 that the application combinator is *applyTU*. The scheme is *full_tdTU*, which means that the functions traverse the whole tree from top to bottom. In line 4, we see that the *default* is an empty list (*constTU []*) and the specific rewrite step is *idn*. The step *idn* returns every term that are of type *Identifier*. Note that the result is a monad, [2], with a list of strings.

Other functions that also incorporate the Strafunski traversal scheme are.

getClassMemberDeclarations :: (Term t, Monad m) => t -> m [ClassMemberDeclaration]

getStaticInitializers :: (Term t, Monad m) => t -> m [StaticInitializer]

getConstructorDeclarations :: (Term t, Monad m) => t -> m [ConstructorDeclaration]

getEmptyClassBodyDeclarations :: (Term t, Monad m) => t -> m [ClassBodyDeclaration]

getClassDeclarations :: (Term t, Monad m) => t -> m [ClassDeclaration]

getInterfaceDeclarations :: (Term t, Monad m) => t -> m [InterfaceDeclaration]

Examples of usage of these functions are found in section 6.2.3. The implementations of these functions are almost identical with the function **getIdentifiers.** The only difference is the rewrite step. We give an example to illustrate this.

```
1    getClassDeclarations cus
2        = applyTU (full_tdTU worker) cus
3              where
4              worker = constTU [] `adhocTU` clssDec
5              clssDec (unit::ClassDeclaration)= return [ unit]
```

Function 23. getClassDeclarations

The real differences between this function and the function **getIdentifiers** are the name of the rewrite step and the type of the term to be returned.

Strafunski uses a Java SDF grammar to generate the Haskell data types. We have introduced new constructions to express differences between two nodes when the generated Haskell data types are not sufficient. We address these constructions in section 6.1 and the constructor *Nothing31* is an example of such construction, and it is found in section 6.2.2 when we discuss method bodies. These constructions are not formally defined in the Java SDF grammar, therefore the Strafunski traversal schemes do not recognize them. We are not able to apply the traversal functions described in this section to any abstract syntax trees that contain such constructors. Only the largest common subtrees might have such constructors and we do not need to traverse them.

We do not show the implementation for the others because they basically are the same. These functions are powerful yet simple and compact, and demonstrate generic programming at its best.

7. Experiment

We have used our tool to compare different versions of a program to measure the tool's performance. We apply series of CVS-versions of two source files of The Eclipse[1] project to our tool and *diff* and compared their outputs. Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. The Eclipse platform currently contains three projects and The Eclipse project is one of them.

The two files that we use are relatively small (64 lines and 214 lines in average), but they have gone through much iteration. Another reason for choosing these files is that Strafunski currently do not support inner classes. It fails to construct abstract syntax trees when applied to such files. The tasks of the two files are to managing property values and to load classes.

We measure the size of the outputs of the two tools when given two versions of a program as arguments. Then we calculate the percentage of the outputs compared to the combined size of the two arguments. These percentages are plotted in the charts in this chapter.

The tool *diff* reports the lines that differ in both versions. Therefore we measure the number of lines in *diff's* output relative to the total number of lines in the two versions. But with our tool we measure the number of nodes of its outputs relative to the total number of nodes of the two abstract syntax trees. By measuring percentage of the outputs, we get a more adequate comparison between the two tools. Otherwise the scale of the output from our tool would be considerable higher than *diff's* outputs. Take for example CVS version 14 and CVS version 15 of the file *AntClassLoader.java*, see section 7.1. The result of comparing CVS version 14 and CVS version 15 is 172 nodes when using our tool. The result when using *diff* is 13 lines. There is a big gap between the numbers 172 and 13. It is unfair to compare the two numbers because the first denotes the number of nodes and the second denotes the number of lines. However, 172 nodes are 10% of the total number of nodes of the two abstract syntax trees that represent CVS version 14 and CVS version 15. And 13 lines are 7% of the total number of lines of CVS version 14 and CVS version 15. Comparing the two percentages gives a better picture of proportions.

The values on the horizontal axis in the charts below represent the different versions of the files being compared. The oldest version has number one. The vertical axis denotes the percentage of the size of the outputs compared to the size of the original files. Note that the measured values depend on two versions of the file. For example in Figure 8, the value at CVS version 4 is about 20 %. This 20 % is calculated by measuring the size of the output of *diff* when it is given CVS version 4 and CVS version 5. The last value in the same chart is calculated using CVS version 21 and CVS version 22, but the last version is not plotted in the chart.

[1] http://eclipse.org

7.1. The results

The first experiment is of the file *PropertyManager.java*[1] and the result is plotted in Figure 7. The size of this file ranges from 169 lines to 287 lines, and the average number of lines that differ from one version to another is 49. All the versions implement a minimum of one interface and at most three.

We see that the two curves, Java AST (our tool) and *diff* are close together most of the time. The only times that there are considerable differences between the two curves are at CVS versions 2, 7 and 19. The average value from Java AST is about 9% and about 12% from *diff*.

The changes from CVS version 2 to CVS version 3 are a few updates in the comments and deleting a few test clauses for catching exceptions. The changes do not affect many lines of codes, but the percentage of outputs nodes compared the total number of nodes, is much higher than the percentage of lines compared to the total number of lines of the two versions.

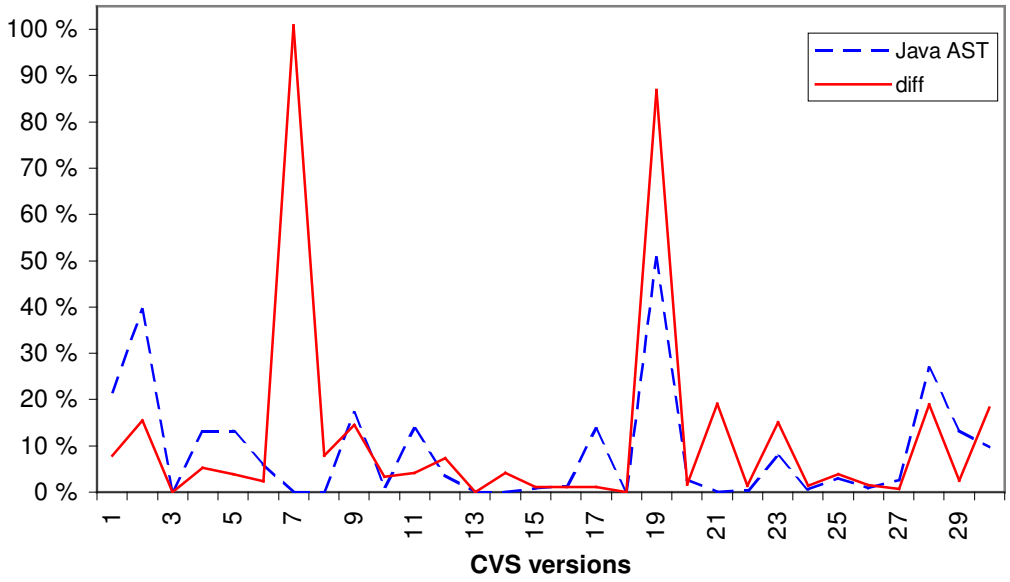


Figure 7. Size of the report of differences for file PropertyManager.java

There are no syntactic differences from CVS version 7 to CVS version 8. The only difference is that CVS version 7 has 175 lines and CVS version 8 has 174 lines. In the eyes of Java AST, the two versions are identical. But somehow *diff* fails to recognize that the two versions are the same. It actually reports that the entire first version differ from the second version. This is the reason for the big gap in CVS version 7.

There is on the other hand done extensive work from CVS version 19 to CVS version 20. Several methods have been extended and in total 54 lines have been added. This is the biggest

[1] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.core.resources/src/org/eclipse/core/internal/properties/PropertyManager.java

change in the series of CVS versions. We see that both tools indicate that there are many changes. But *diff's* measured output is much higher than the measured output from Java AST.

Java AST reports nothing at the CVS versions 3, 7, 8, 13, 14, 18, 21 and 22 because there is either no differences or the changes are comments. The tool *diff* also reports the same in CVS versions 3, 13 and 18. The average size of the outputs from Java AST is a few percentages below *diff's*. But this is mostly due to the unexplainable and strange result from *diff* in CVS version 7. If we set *diff's* result in CVS version 7 to 0%, which reflects the actual changes much better, then *diff's* outputs is about 8% in average. This is about the same as Java AST's average.

A second experiment is done to see if there is trend in the results. The file we compared this time is *AntClassLoader*.java[1] and we plotted the result in Figure 8. The size of this file is smaller, and it ranges from 46 lines to 96 lines. The average number of lines that differ from one version to another is 20. The versions do not implement any interfaces, but they have one super class. Also this time the curves stay close together most of the time. We see that biggest gaps are in CVS versions 4, 7 and 16. The size of the outputs of Java AST is 9% in average and 16% in average for *diff's* outputs.

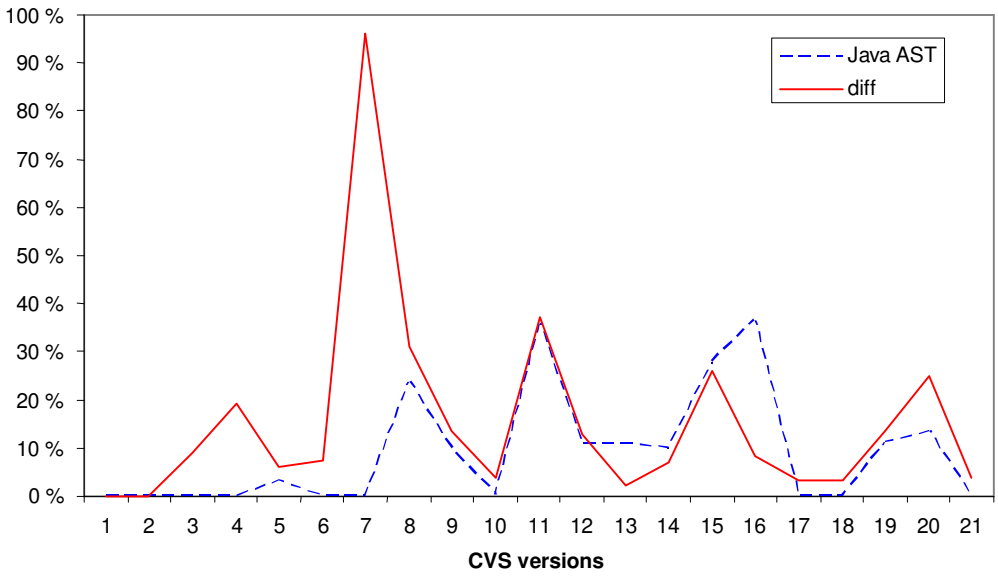


Figure 8. Size of the reports of differences for file AntClassLoader.java

The changes at the CVS versions 1, 2, 3, 4, 6, 7, 17 and 21 are only in the comments. No syntactic changes are done. Therefore Java AST does not report any differences. It seems that Java AST in Figure 8 at CVS versions 10 and 17 also report nothing. The truth is that one line has changed in the source code from CVS version 10 to CVS version 11 and from CVS version 17 to CVS version 18. The result in both occasions is only 2 nodes that differ from a total of 1052 nodes and 1096 nodes. The results are too small and impossible to see in the chart.

[1] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ant.core/src/org/eclipse/ant/internal/core/AntClassLoader.java?rev=1.20&content-type=text/vnd.viewcvs-markup

Again we see the huge gap between the two outputs in CVS version 7. The code in CVS version 8 has been formatted since CVS version 7. The formatting does not have any syntactic consequences; therefore Java AST does not report anything. The tool *diff* in the other hand fails to recognize the two versions as equivalent and report that 96% of the first version does not match the second version.

The most changes done in the series of versions are from CVS version 11 to CVS version 12, which are reflected by the high percentage of the two outputs. The difference between the two outputs is very small and in favor of Java AST.

The biggest gap between the two outputs that favor *diff* is found at CVS version 16. There are not many lines that have been changed from CVS version 16 to CVS version 17, but most significant to Java AST is that a method name has been changed. Java AST compares the methods signatures to match a method in the first version with the same method in the second version. And since the signature is not the same, then Java AST considers them as two different methods. It then reports that the method in first version is deleted and the method in the second version is added. The two bodies are identical and therefore *diff* reports in this case that only one line is different.

7.2. Comments

The only solid conclusion that we can draw from these experiments is that *diff* sometimes fails miserably when applied to programs. The result at CVS version 7 in Figure 8 shows that *diff* easily get confused when the code is reformatted and we are still puzzled by the result at CVS version 7 in Figure 7.

We see also that comments have no effect on Java AST. It reports in total 8 times to *diff's* three times that there are no changes in Figure 7. And Java AST also reports 8 times in Figure 8 that there are no syntactic changes, while *diff* only reports the same on two occasions.

Overall we see that the outputs from Java AST are a little smaller than the outputs from *diff*. As mentioned earlier, *diff* sometimes fails totally, but most times it manages quit well. When the changes are small compared to the original file's size, it does not matter that some lines are irrelevant. It is when the changes are large that we see the potential for Java AST. As in CVS version 19 in Figure 7, the output of *diff* is much bigger than the outputs of Java AST. But Java AST also produces large outputs. In CVS version 16 in Figure 8, the Java AST output is several times lager than *diff's* output. This is due to the tools different approach for matching methods.

We must also mention that these experiments do not measures the performance of Java AST accurately since the outputs are not yet formatted. We use the raw outputs from Java AST and it is unfair to compare those with the outputs from *diff*. Formatting the outputs from Java AST gives text lines and code fragment, similar to *diff's* outputs. For some situations, such as comparing statements in method bodies, the two outputs should be very similar in size and content, see section 3.4.4. The comparison between those outputs is a better way to measure the performance of the two tools. That comparison is not in the scope of this thesis.

8. Conclusions and future work

We have improved and extended Yang's algorithms to fit our purpose, which is to compare Java programs. But the fundamental algorithms do not only work with Java, it can easily be adapted to other programming languages. We do not introduce and adapt to Java until the final comparison functions. All the fundamental algorithms and functions are designed to compare abstract syntax trees and not specific to Java.

Yang's algorithms only compute the length of the longest common subsequence and the number of nodes in the largest common subtree. The algorithms compute these values with help of matrices. Our goal is to find the actual longest common subsequence and the actual largest common subtree. We have therefore modified Yang's algorithms to return the whole matrices instead, because the key to find subsequences and subtrees lay in the matrices. We give also algorithms for traversing the matrices and extracting the subsequences and subtrees from them. These algorithms are then combined with Yang's modified algorithms and the result is algorithms that serve our purpose; finding longest common subsequences and largest common subtrees.

Our tool has a different approach for comparing Java programs than *diff* or *JDiff*. This new approach gives more useful and more precise outputs. Good comparisons tools for Java programs are much needed because Java is currently one of the most popular programming languages. It is our conclusion that we laid the ground for the development of a useful tool with much potential.

8.1. Critique and areas for improvement

The fundamental algorithms in this thesis are small and compact, which means that there is much room for improving the efficiency. For example, the algorithm for comparing two lists of unordered nodes, algorithm **Tree matching unordered**, does not move on when it finds a match. It keeps matching the same node until it has matched it with the entire second list. This is of course not necessary. The function **orderedMaximumMatching** finds the longest common subsequence in two steps; it generates a matrix and then traverses it. We have discussed that it is not possible to do this in one step in Haskell. Perhaps it is possible to take advantage of Haskell's lazy evaluation to improve the efficiency; only compute the elements in matrix M that are needed.

The main area for improvement is the implementation of the tool presented in this thesis. We are novices with Haskell programming and the implementation reflects that. Much work can be done to compact the code since we were not concerned with that during implementation. Our solution is to give each of the data types, which are generated by Strafunski, a function of its own. But it is possible to groups several data types together and to make a single function that applies to all of them. Many data types are similar and do not need a function with a specific comparison rule.

We have also introduced a series of constructors used when we cannot express differences without violating the grammar. These constructors have been added without being defined in the Java SDF grammar. Strafunski generates Haskell data types for the Java language based

on this grammar. The consequence of this is that we are not able to traverse abstract syntax trees that contain these new constructors with Strafunski's traversal schemes because they do not recognize the new constructors. But only the largest common subtrees might have these constructors. We do not in this thesis need to traverse the largest common subtrees using Strafunski traversal schemes. But if Strafunski should some day support pretty-printing, then our largest common subtrees are not syntactically correct. Defining the constructors in the SDF grammar will solve these problems.

We set out in this thesis to produce a tool that improves the quality of the difference reports. The judgment of the quality is a subjective matter and it should be the subject of empirical testing. We focus much in this thesis on the length of the common subsequences and the size of the largest common subtrees, but these values are not good indicators of quality in our case. The shortest report of differences is not always the best. Consider the Java example 3 in section 3.4.3, where a class removes an interface of one version to another, but it keeps all the inherited methods of the interface. The shortest report is to only say that the class has removed the interface. But the best report of differences is to say further that the mentioned methods are now declared methods in the second version.

The experiment in this thesis only focuses on the size of the outputs from our tool and *diff*. The two outputs are not in the same scale and it is unfair to compare the two. The only way to conclude that we have improved the quality of the reports of differences is case studies and interviews. Real programmers using our tool in their work are the best judges. The feedbacks from them must be the base of a final conclusion.

8.2. Future work

Future work includes improving the difference reports regarding for other aspects of Java, such as concurrency, abstract classes etc. Other work includes using an updated version of the SDF grammar for Java 1.5 (J2SE 5.0) so Strafunski can create abstract syntax trees from programs that have inner class constructs.

We have discussed the finding of multiple solutions. The work in this thesis does not discard that option, and it is possible to extend our work to pursue that line. Last but not least, a reader-friendly report must be generated for the ease of use.

8.3. Acknowledgments

We like to thank Joost Wisser for helping us install and supporting us regarding Strafunski. Strafunski is a great tool and it simplified our work a lot. Without it, this thesis would have taken longer to complete or be less comprehensive than it is now. Most important of all, we would like to thank Bjarte M. Østvold. Thank you for all the help, support and guidance. Your consistency about weekly meetings was the driving force that made it possible to finish this

thesis in the time that it did. And your thorough comments during the entire period of writing have improved this thesis tremendously.

Appendix A. Reference list

[1] Wuu Yang, "Identifying Syntactic Differences Between Two Programs", Software – Practice and Experience, vol 21 no 7 pages 739-755, 1991.

[2] Simon Peyton Jones, "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", Engineering theories of software construction, ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, pages 47-96, 2001.

[3] Peter Sestoft, "Java Precisely", version 1.05 of 2000-11-23, IT University of Copenhagen, Denmark and Royal Veterinary and Agricultural University, Copenhagen, Denmark.

[4] Carl Ghezzi and Mehdi Jazayeri, "Programming Language Concepts", third edition, John Wiley & Sons, Inc, 1998.

[5] Simon Thomson, "The Craft of Functional Programming", Addison Wesley Longman, ISBN 0-201-40357-9, 1996

[6] Joost Visser, "Visitor Combination and Traversal Control", Conference on Object-Oriented Programming Systems, Languages and Applications, pages 270-282, 2001

[7] Tobias Kuipers and Joost Visser, "Object-Orientated Tree Traversal with JJForester", Electronic Notes in Theoretical Computer Science, vol 44 no 2, pages 1-25, 2001.

[8] Jason T.L Wang and Kaizhong Zhang, "Finding Similar Consensus Between Trees: an Algorithm and a distance Hierarchy", Pattern Recognition, vol 34 no 1, pages 127-137 2001.

[9] Ralf Lämmel and Joost Visser, "Typed Combinators for Generic Traversal", Practical Aspects of Declarative Languages, 4th International Symposium, pages 137-154, 2002.

[10] Sun Wu, Udi Manber, Gene Myers and Webb Miller, "An O(NP) Sequence Comparison Algorithm", Information Processing Letters, vol 35 no 6, pages 317-323, 1990.

[11] Christian Charras and Thierry Lecroq, University of Rouen, web page: http://www-igm.univ-mlv.fr/~lecroq/string/index.html

[12] Somesh Jha, Jens Palsberg and Tian Zhao, "Efficient Type Matching", Foundations of Software Science and Computation Structure, pages 187-204, 2002

[13] David Binkley, Rob Capellini, L. Ross Raszewski and Christopher Smith, "An Implementation of and Experiment with Semantic Differencing", International Conference on Software Maintenance, pages 82-91, 2001.

[14] Tom Mens, "State-of-the-Art Survey on Software Merging", IEEE Transaction on Software Engineering, vol. 28, no.5, 2002

[15] Ralf Lämmel and Joost Visser, "A Strafunski Application Letter", Practical Aspects of Declarative Languages, 5th International Symposium, pages 357-375, 2003.

Appendix B. Source code

File MatchingWStr.hs task is to compare the two abstract syntax trees.

```
module MatchingWStr where

import Java
import JavaATermConvertibleInstances
import JavaTermInstances
import JavaChaseImports
import ChaseImports
import StrategyLib
import ATermLib
import System
import DTDJavaMetrics
import DTDJavaMetricsTermInstances
import Text.XML.HaXml.Xml2Haskell
import IO
import Monad
import SequenceMatching


getIdentifier ::(Term t, Monad m) => t -> m [String]
getIdentifier cus = applyTU (full_tdTU id) cus
                    where
                    id = constTU [] `adhocTU` idn
                    idn (identifier::Identifier) = return [show identifier]

getFieldIdentifier ::(Term t, Monad m) => t -> m [String]
getFieldIdentifier cus = applyTU (full_tdTU id) cus
                    where
                    id = constTU [] `adhocTU` idn
                    idn (Identifier1 identifier) = return [show identifier]

getMethIdentifier ::(Term t, Monad m) => t -> m [String]
getMethIdentifier cus = applyTU (full_tdTU id) cus
                    where
                    id = constTU [] `adhocTU` idn
                    idn (Comma2 identifier _) = return [show identifier]

getFirstId :: (Term t) => t -> String
getFirstId cus
   | id == [] = ""
   | otherwise = head id
                     where
                     [id] = getIdentifier cus


getClassMemberDeclarations :: (Term t, Monad m) => t -> m [ClassMemberDeclaration]
getClassMemberDeclarations cus
                     = applyTU (full_tdTU worker) cus
                       where
                       worker = constTU [] `adhocTU` clssDec
                       clssDec (unit::ClassMemberDeclaration)= return [unit]


getStaticInitializers :: (Term t, Monad m) => t -> m [StaticInitializer]
getStaticInitializers cus
```

```
                    = applyTU (full_tdTU worker) cus
                      where
                      worker = constTU [] `adhocTU` clssDec
                      clssDec (a@(Static1 _)) = return [a]


getConstructorDeclarations :: (Term t, Monad m) => t -> m [ConstructorDeclaration]
getConstructorDeclarations cus
                    = applyTU (full_tdTU worker) cus
                      where
                      worker = constTU [] `adhocTU` clssDec
                      clssDec (a@(Modifier_s_ConstructorDeclarator_Throws_opt_ConstructorBody _ _ _ _)) =
return [a]

getEmptyClassBodyDeclarations :: (Term t, Monad m) => t -> m [ClassBodyDeclaration]
getEmptyClassBodyDeclarations cus
                    = applyTU (full_tdTU worker) cus
                      where
                      worker = constTU [] `adhocTU` clssDec
                      clssDec (EmptyClassBodyDeclaration) = return [EmptyClassBodyDeclaration]

getClassDeclarations :: (Term t, Monad m) => t -> m [ClassDeclaration]
getClassDeclarations cus
                    = applyTU (full_tdTU worker) cus
                      where
                      worker = constTU [] `adhocTU` clssDec
                      clssDec (unit::ClassDeclaration)= return [ unit]

getInterfaceDeclarations :: (Term t, Monad m) => t -> m [InterfaceDeclaration]
getInterfaceDeclarations cus
                    = applyTU (full_tdTU worker) cus
                      where
                      worker = constTU [] `adhocTU` clssDec
                      clssDec (unit::InterfaceDeclaration)= return [unit]


getTypeFromFP :: (Term t, Monad m) => t -> m [String]
getTypeFromFP cus
              = applyTU (full_tdTU worker) cus
                    where
                    worker = constTU [] `adhocTU` tp
                    tp (Type_VariableDeclaratorId tps _) = return [show tps]


getMethodName ::(Term t) => t -> String
getMethodName cus = head id
                     where
                     [id] = getMethIdentifier cus


compilationUnit :: CompilationUnit ->CompilationUnit ->CompilationUnit
compilationUnit (CU pdA impsA tpsA)(CU pdB impsB tpsB)
   = CU (packageDeclaration pdA pdB) (importDeclaration impsA impsB) (typeDeclaration tpsA tpsB)

packageDeclaration :: Maybe PackageDeclaration ->Maybe PackageDeclaration ->Maybe PackageDeclaration
packageDeclaration pkdA pdkB
   | pdkA == pdkB = pdkA
   | otherwise = Nothing
```

```
importDeclaration :: [ImportDeclaration] -> [ImportDeclaration] -> [ImportDeclaration]
importDeclaration a b = unorderedMaximumMatching a b b


typeDeclaration :: [TypeDeclaration] ->[TypeDeclaration] ->[TypeDeclaration]
typeDeclaration a b = typeDeclarationList a b b
typeDeclarationList :: [TypeDeclaration] ->[TypeDeclaration] ->[TypeDeclaration]->[TypeDeclaration]
typeDeclarationList (_:tailA) [] orgB = typeDeclarationList tailA orgB orgB
typeDeclarationList [] _ _ = []
typeDeclarationList a@((ClassTypeDeclaration cdA):tailA)((ClassTypeDeclaration cdB):tailB) orgB
   | idA == idB = [ClassTypeDeclaration (compareClassDeclaration cdA cdB)] ++ typeDeclarationList tailA
orgB orgB
   | otherwise = typeDeclarationList a tailB orgB
                  where
                  idA = getFirstId cdA
                  idB = getFirstId cdB
typeDeclarationList a@((InterfaceDeclaration ifdA):tailA)((InterfaceDeclaration ifdB):tailB) orgB
   | idA == idB = [InterfaceDeclaration (compareInterfaceDeclaration ifdA ifdB )]++ typeDeclarationList tailA
orgB orgB
   | otherwise = typeDeclarationList a tailB orgB
                  where
                  idA = getFirstId ifdA
                  idB = getFirstId ifdB
typeDeclarationList (Semicolon5:tailA) (Semicolon5:tailB) orgB = [Semicolon5] ++ typeDeclarationList tailA
orgB orgB
typeDeclarationList a (hd:tailB) orgB = typeDeclarationList a tailB orgB


super :: Maybe Super ->Maybe Super ->Maybe Super
super (Nothing) _ = Nothing
super _ (Nothing) = Nothing
super (Just (Extends clstpA)) (Just (Extends clstpB)) = Just (Extends (classType clstpA clstpB))

interfaces :: Maybe Interfaces ->Maybe Interfaces ->Maybe Interfaces
interfaces (Nothing) _ = Nothing
interfaces _ (Nothing) = Nothing
interfaces (Just (Implements_comma itpA))(Just (Implements_comma itpB))= Just (Implements_comma
(interfaceType itpA itpB))


interfaceType :: [InterfaceType]->[InterfaceType]->[InterfaceType]
interfaceType itpA itpB = interfaceTypeList itpA itpB itpB
interfaceTypeList :: [InterfaceType]->[InterfaceType]->[InterfaceType]->[InterfaceType]
interfaceTypeList [] _ _ = []
interfaceTypeList (_:tailA) [] orgB = interfaceTypeList tailA orgB orgB
interfaceTypeList a@(headA:tailA) (headB:tailB) orgB
   | headA == headB = [headA]++interfaceTypeList tailA orgB orgB
   | otherwise = interfaceTypeList a tailB orgB


classBody :: ClassBody ->ClassBody ->ClassBody
classBody (ClassBodyDeclaration_s clsbdsA)(ClassBodyDeclaration_s clsbdsB)
   = ClassBodyDeclaration_s (clms++st++cs++es++cl1++cl2)
     where
     clms = classMemberDeclaration clmsA clmsB
     st = staticInitializer stA stB
     cs = constructorDeclaration csA csB
     es = []
     --es = emptyClassBodyDeclaration esA esB
     cl1 = classDeclaration cl1A cl1B
```

```
    cl2 = interfaceDeclaration cl2A cl2B
    [clmsA] = getClassMemberDeclarations clsbdsA
    [clmsB] = getClassMemberDeclarations clsbdsB
    [stA] = getStaticInitializers clsbdsA
    [stB] = getStaticInitializers clsbdsB
    [csA] = getConstructorDeclarations clsbdsA
    [csB] = getConstructorDeclarations clsbdsB
    [esA] = getEmptyClassBodyDeclarations clsbdsA
    [esB] = getEmptyClassBodyDeclarations clsbdsB
    [cl1A] = getClassDeclarations clsbdsA
    [cl1B] = getClassDeclarations clsbdsB
    [cl2A] = getInterfaceDeclarations clsbdsA
    [cl2B] = getInterfaceDeclarations clsbdsA


staticInitializer :: [StaticInitializer]->[StaticInitializer]->[ClassBodyDeclaration]
staticInitializer stA stB = staticInitializerList stA stB stB
staticInitializerList :: [StaticInitializer]->[StaticInitializer]->[StaticInitializer]->[ClassBodyDeclaration]
staticInitializerList (head:tail) [] orgB = staticInitializerList tail orgB orgB
staticInitializerList [] _ _ = []
staticInitializerList a@((Static1 blA):tailA) ((Static1 blB):tailB) orgB
  | blA == blB = [StaticInitializer (Static1 blA)]++staticInitializerList tailA orgB orgB
  | otherwise = staticInitializerList a tailB orgB


constructorDeclaration :: [ConstructorDeclaration]->[ConstructorDeclaration]->[ClassBodyDeclaration]
constructorDeclaration ctA ctB = constructorDeclarationList ctA ctB ctB
constructorDeclarationList :: [ConstructorDeclaration]->[ConstructorDeclaration]->[ConstructorDeclaration]
                              ->[ClassBodyDeclaration]
constructorDeclarationList (head:tail) [] orgB = constructorDeclarationList tail orgB orgB
constructorDeclarationList [] _ _ = []
constructorDeclarationList a@((Modifier_s_ConstructorDeclarator_Throws_opt_ConstructorBody modsA cdA
thA bdA):tailA)
                           ((Modifier_s_ConstructorDeclarator_Throws_opt_ConstructorBody modsB cdB
thB bdB):tailB) orgB
  | cdA == cdB = [ConstructorDeclaration (Modifier_s_ConstructorDeclarator_Throws_opt_ConstructorBody
cMods cdA cTh cBd)]
                  ++constructorDeclarationList tailA orgB orgB
  | otherwise = constructorDeclarationList a tailB orgB
                where
                cMods = modifier modsA modsB
                cTh = throws thA thB
                cBd = constructorBody bdA bdB


constructorBody :: ConstructorBody -> ConstructorBody -> ConstructorBody
constructorBody (ExplicitConstructorInvocation_opt_BlockStatement_s exciA blcksA)
                 (ExplicitConstructorInvocation_opt_BlockStatement_s exciB blcksB)
  = ExplicitConstructorInvocation_opt_BlockStatement_s (explicitConstructorInvocation exciA exciA)
    (blockStatement blcksA blcksB)


explicitConstructorInvocation ::Maybe ExplicitConstructorInvocation ->Maybe ExplicitConstructorInvocation
                                ->Maybe ExplicitConstructorInvocation
explicitConstructorInvocation Nothing _ = Nothing
explicitConstructorInvocation _ Nothing = Nothing
explicitConstructorInvocation (Just (This_comma_semicolon _)) (Just (Super_comma_semicolon _)) = Nothing
explicitConstructorInvocation (Just (Super_comma_semicolon _)) (Just (This_comma_semicolon _)) = Nothing
explicitConstructorInvocation (Just (This_comma_semicolon exsA)) (Just (This_comma_semicolon exsB))
```

```
   = Just (This_comma_semicolon (expressionList exsA exsB))
explicitConstructorInvocation (Just (Super_comma_semicolon spA))(Just (Super_comma_semicolon spB))
   = Just (Super_comma_semicolon (expressionList spA spB))


expressionList :: [Expression]->[Expression]->[Expression]
expressionList exA exB = maximumMatching exA exB


emptyClassBodyDeclaration :: [ClassBodyDeclaration]->[ClassBodyDeclaration]->[ClassBodyDeclaration]
emptyClassBodyDeclaration a b
   | (length a) > (length b) = b
   | otherwise = a


compareClassDeclaration :: ClassDeclaration ->ClassDeclaration ->ClassDeclaration
compareClassDeclaration (Class1 mdsA idA spA infA bdA)(Class1 mdsB idB spB infB bdB)
   | idA == idB = Class1 (modifier mdsA mdsB) idA (super spA spB) (interfaces infA infB) (classBody bdA
bdB)
   | otherwise = Nothing32


classDeclaration :: [ClassDeclaration]->[ClassDeclaration]->[ClassBodyDeclaration]
classDeclaration cdA cdB = classDeclarationList cdA cdB cdB
classDeclarationList :: [ClassDeclaration]->[ClassDeclaration]->[ClassDeclaration]->[ClassBodyDeclaration]
classDeclarationList (head:tail) [] orgB = classDeclarationList tail orgB orgB
classDeclarationList [] _ _ = []
classDeclarationList a@(cdA:tailA) (cdB:tailB) orgB
   | cp == Nothing32 = classDeclarationList a tailB orgB
   | otherwise = [ClassDeclaration1 cp] ++ classDeclarationList tailA orgB orgB
                  where
                  cp = compareClassDeclaration cdA cdB


interfaceDeclaration :: [InterfaceDeclaration]->[InterfaceDeclaration]->[ClassBodyDeclaration]
interfaceDeclaration ifA ifB = interfaceDeclarationList ifA ifB ifB
interfaceDeclarationList :: [InterfaceDeclaration]->[InterfaceDeclaration]->[InterfaceDeclaration]-
>[ClassBodyDeclaration]
interfaceDeclarationList (head:tail) [] orgB = interfaceDeclarationList tail orgB orgB
interfaceDeclarationList [] _ _ = []
interfaceDeclarationList a@(a_@(Interface _ idA _ _):tailA) (b_@(Interface _ idB _ _):tailB) orgB
   | idA == idB = [ClassDeclaration2 (compareInterfaceDeclaration a_ b_)] ++ interfaceDeclarationList tailA
orgB orgB
   | otherwise = interfaceDeclarationList a tailB orgB


compareInterfaceDeclaration :: InterfaceDeclaration -> InterfaceDeclaration -> InterfaceDeclaration
compareInterfaceDeclaration (Interface modsA idA exiA bdA)(Interface modsB idB exiB bdB)
   = Interface (modifier modsA modsB) idA (extendsInterfaces exiA exiB) (interfaceBody bdA bdB)


extendsInterfaces :: Maybe ExtendsInterfaces -> Maybe ExtendsInterfaces -> Maybe ExtendsInterfaces
extendsInterfaces Nothing _ = Nothing
extendsInterfaces _ Nothing = Nothing
extendsInterfaces (Just(Extends_comma iftpA))(Just(Extends_comma iftpB)) = Just(Extends_comma
(interfaceType iftpA iftpB))


interfaceBody :: InterfaceBody -> InterfaceBody -> InterfaceBody
interfaceBody (InterfaceMemberDeclaration_s ifmdA) (InterfaceMemberDeclaration_s ifmdB)
```

```
   = InterfaceMemberDeclaration_s (interfaceMemberDeclaration ifmdA ifmdB)


interfaceMemberDeclaration :: [InterfaceMemberDeclaration]->[InterfaceMemberDeclaration]-
>[InterfaceMemberDeclaration]
interfaceMemberDeclaration a b = interfaceMemberDeclarationList a b b
interfaceMemberDeclarationList :: [InterfaceMemberDeclaration]->[InterfaceMemberDeclaration]-
>[InterfaceMemberDeclaration]
                               ->[InterfaceMemberDeclaration]
interfaceMemberDeclarationList (head:tail) [] orgB = interfaceMemberDeclarationList tail orgB orgB
interfaceMemberDeclarationList [] _ _ = []
interfaceMemberDeclarationList a@((ConstantDeclaration cdA):tailA) ((ConstantDeclaration cdB):tailB) orgB
   | idA == idB = [ConstantDeclaration (constantDeclaration cdA cdB)]++ interfaceMemberDeclarationList
tailA orgB orgB
   | otherwise = interfaceMemberDeclarationList a tailB orgB
                 where
                 idA = head (getFieldIdentifier cdA)
                 idB = head (getFieldIdentifier cdB)
interfaceMemberDeclarationList a@((AbstractMethodDeclaration amA):tailA)((AbstractMethodDeclaration
amB):tailB) orgB
   | idA == idB = [AbstractMethodDeclaration (abstractMethodDeclaration amA amB)]
                  ++ interfaceMemberDeclarationList tailA orgB orgB
   | otherwise = interfaceMemberDeclarationList a tailB orgB
                 where
                 idA = head (getMethIdentifier amA)
                 idB = head (getMethIdentifier amB)
interfaceMemberDeclarationList a (head:tail) orgB  = interfaceMemberDeclarationList a tail orgB


constantDeclaration :: ConstantDeclaration ->ConstantDeclaration ->ConstantDeclaration
constantDeclaration (FieldDeclaration fdA)(FieldDeclaration fdB) = FieldDeclaration (fieldDeclaration fdA fdB)


abstractMethodDeclaration :: AbstractMethodDeclaration ->AbstractMethodDeclaration -
>AbstractMethodDeclaration
abstractMethodDeclaration (Semicolon4 mhA)(Semicolon4 mhB) = Semicolon4 (methodHeader mhA mhB)

classMemberDeclaration :: [ClassMemberDeclaration] -> [ClassMemberDeclaration] -> [ClassBodyDeclaration]
classMemberDeclaration clsmbA clsmbB = classMemberDeclarationList clsmbA clsmbB clsmbB

classMemberDeclarationList :: [ClassMemberDeclaration] -> [ClassMemberDeclaration] ->
[ClassMemberDeclaration] ->[ClassBodyDeclaration]
classMemberDeclarationList ((FieldDeclaration1 fdA):tailA) [] orgB = classMemberDeclarationList tailA orgB
orgB
classMemberDeclarationList [] _ _ = []
classMemberDeclarationList a@((FieldDeclaration1 fdA:tailA)) ((MethodDeclaration _):tailB) orgB
   = classMemberDeclarationList a tailB orgB
classMemberDeclarationList a@((FieldDeclaration1 fdA):tailA) ((FieldDeclaration1 fdB):tailB) orgB
   | fdAName == fdBName = [ClassMemberDeclaration(FieldDeclaration1 (fieldDeclaration fdA fdB))]
                          ++classMemberDeclarationList tailA orgB orgB
   | otherwise = classMemberDeclarationList a tailB orgB
                 where
                 fdAName = head (getFieldIdentifier fdA)
                 fdBName = head (getFieldIdentifier fdB)


classMemberDeclarationList ((MethodDeclaration mdA):tailA) [] orgB = classMemberDeclarationList tailA
orgB orgB
classMemberDeclarationList a@((MethodDeclaration mdA):tailA) ((FieldDeclaration1 _):tailB) orgB
```

```
   = classMemberDeclarationList a tailB orgB
classMemberDeclarationList a@((MethodDeclaration mdA):tailA) ((MethodDeclaration mdB):tailB) orgB
   | compareMethodSign mdA mdB = [ClassMemberDeclaration (MethodDeclaration (methodDeclaration mdA
mdB))]
                                 ++classMemberDeclarationList tailA orgB orgB
   | otherwise = classMemberDeclarationList a tailB orgB



compareMethodSign :: MethodDeclaration -> MethodDeclaration -> Bool
compareMethodSign mdA mdB
   | chckA || chckB  = False
   | mdAName == mdBName  &&  mdAfp == mdBfp = True
   | otherwise = False
                  where
                  mdAName = getMethodName mdA
                  mdBName = getMethodName mdB
                  [mdAfp] = getTypeFromFP mdA
                  [mdBfp] = getTypeFromFP mdB
                  chckA = isInheritedMethod "InheritedMethod:" mdAName
                  chckB = isInheritedMethod "InheritedMethod:" mdBName


isInheritedMethod :: String -> String -> Bool
isInheritedMethod a [] = False
isInheritedMethod [] _ = True
isInheritedMethod (hdA:tlA) (hdB:tlB)
   | hdA == hdB = isInheritedMethod tlA tlB
   | otherwise = False


fieldDeclaration :: FieldDeclaration -> FieldDeclaration -> FieldDeclaration
fieldDeclaration (Comma_semicolon modsA typeA varDecA) (Comma_semicolon modsB typeB varDecB)
   = Comma_semicolon (modifier modsA modsB) (typeMatching typeA typeB) (variableDeclarator varDecA
varDecA)


modifier :: [Modifier] -> [Modifier] -> [Modifier]
modifier modsA modsB = modifierList modsA modsB modsB
modifierList :: [Modifier] -> [Modifier] -> [Modifier]-> [Modifier]
modifierList [] _ _ = []
modifierList (_:tailA) [] orgB = modifierList tailA orgB orgB
modifierList a@(headA:tailA) (headB:tailB) orgB
   | headA == headB = [headA]++modifierList tailA orgB orgB
   | otherwise = modifierList a tailB orgB


typeMatching :: Type -> Type -> Type
typeMatching (PrimitiveType prtpA) (PrimitiveType prtpB) = PrimitiveType (primitiveType prtpA prtpB)
typeMatching (ReferenceType rftpA) (ReferenceType rftpB) = ReferenceType (referenceType rftpA rftpB)
typeMatching _ _ = Nothing1


primitiveType :: PrimitiveType -> PrimitiveType -> PrimitiveType
primitiveType (NumericType nmtpA) (NumericType nmtpB) = NumericType (numericType nmtpA nmtpB)
primitiveType (Boolean) (Boolean) = Boolean
primitiveType _ _ = Nothing2


numericType :: NumericType -> NumericType -> NumericType
```

```
numericType (IntegralType intpA) (IntegralType intpB)
            | (show intpA) == (show intpB) = IntegralType intpA
            | otherwise = IntegralType Nothing4
numericType (FloatingPointType flpttpA) (FloatingPointType flpttpB)
            | (show flpttpA) == (show flpttpB) = FloatingPointType flpttpA
            | otherwise = FloatingPointType Nothing5
numericType _ _ = Nothing3


referenceType :: ReferenceType -> ReferenceType -> ReferenceType
referenceType (ClassOrInterfaceType clsintfctpA) (ClassOrInterfaceType clsintfctpB)
  = ClassOrInterfaceType (classOrInterfaceType clsintfctpA clsintfctpB)
referenceType (ArrayType arrtpA) (ArrayType arrtpB) = ArrayType (arrayType arrtpA arrtpB)
referenceType _ _ = Nothing6


classOrInterfaceType :: ClassOrInterfaceType -> ClassOrInterfaceType -> ClassOrInterfaceType
classOrInterfaceType (Name nameA) (Name nameB)
                     | nameA == nameB = Name nameA
                     | otherwise = Nothing7


arrayType ::  ArrayType -> ArrayType -> ArrayType
arrayType (PrimitiveType1 prtpA) (PrimitiveType1 prtpB) = PrimitiveType1 (primitiveType prtpA prtpB)
arrayType (ArrayType0 nameA) (ArrayType0 nameB) = ArrayType0 (name nameA nameB)
arrayType (ArrayType1 arrtpA) (ArrayType1 arrtpB) = ArrayType1 (arrayType arrtpA arrtpB)
arrayType _ _ = Nothing8


name :: Name -> Name -> Name
name (Identifier_p idsA) (Identifier_p idsB) = Identifier_p commonIdentifiers
                                               where
                                               commonIdentifiers = maximumMatching idsA idsB
name (Class idsclsA) (Class idsclsB) = Class commonIdentifiers1
                                       where
                                       commonIdentifiers1 = maximumMatching idsclsA idsclsB
name _ _ = Nothing9


variableDeclarator :: [VariableDeclarator] ->[VariableDeclarator] ->[VariableDeclarator]
variableDeclarator vrdecA vrdecB =  matchingVariableDec vrdecA vrdecB vrdecB


matchingVariableDec :: [VariableDeclarator] ->[VariableDeclarator] -> [VariableDeclarator]-
>[VariableDeclarator]
matchingVariableDec (vrdecA:tailA) [] orgB = matchingVariableDec tailA orgB orgB
matchingVariableDec [] _ _ = []
matchingVariableDec a@(vrdecA:tailA) (vrdecB:tailB) orgB
                     | idA == idB = [matchingVariableDec1 vrdecA vrdecB]++ matchingVariableDec tailA orgB
orgB
                     | otherwise = matchingVariableDec a tailB orgB
                                   where
                                   idA = getFirstId vrdecA
                                   idB = getFirstId vrdecB


matchingVariableDec1 :: VariableDeclarator -> VariableDeclarator ->VariableDeclarator
matchingVariableDec1 (VariableDeclaratorId vrdcidA) (VariableDeclaratorId vrdcidB)
  = VariableDeclaratorId (variableDeclaratorId vrdcidA vrdcidB)
```

```
matchingVariableDec1 (Equal vrdcidA vrinitA) (Equal vrdcidB vrinitB)
   = Equal (variableDeclaratorId vrdcidA vrdcidB) (variableInitializer vrinitA vrinitB)
matchingVariableDec1 (Equal vrdcA _) (VariableDeclaratorId vrdcB) = Nothing10 (variableDeclaratorId vrdcA
vrdcB)
matchingVariableDec1 (VariableDeclaratorId vrdcA) (Equal vrdcB _) = Nothing10 (variableDeclaratorId vrdcA
vrdcB)


variableDeclaratorId :: VariableDeclaratorId ->VariableDeclaratorId ->VariableDeclaratorId
variableDeclaratorId (Identifier1 idA) (Identifier1 idB)
   | idA == idB = Identifier1 idA
   | otherwise = Identifier1 "No match"
variableDeclaratorId (VariableDeclaratorId1 vrdcidA) (VariableDeclaratorId1 vrdcidB)
   = VariableDeclaratorId1 (variableDeclaratorId vrdcidA vrdcidB)
variableDeclaratorId (Identifier1 idA) (VariableDeclaratorId1 _) = Nothing11 idA
variableDeclaratorId (VariableDeclaratorId1 _) (Identifier1 idB) = Nothing11 idB


variableInitializer :: VariableInitializer -> VariableInitializer -> VariableInitializer
variableInitializer (Expression xprA) (Expression xprB) = Expression (expression xprA xprB)
variableInitializer (ArrayInitializer arinitA)(ArrayInitializer arinitB) = ArrayInitializer (arrayInitializer arinitA
arinitB)
variableInitializer _ _ = Nothing26


expression :: Expression ->Expression ->Expression
expression (Primary prmA) (Primary prmB) = Primary (primary prmA prmB)
expression (Plus xprA) (Plus xprB) = Plus (expression xprA xprB)
expression (Minus xprA) (Minus xprB) = Minus (expression xprA xprB)
expression (Tilde xprA) (Tilde xprB) = Tilde (expression xprA xprB)
expression (Not xprA) (Not xprB) = Not (expression xprA xprB)
expression (PrimitiveType_Dim_s_Expression prmtpA dimA xprA)(PrimitiveType_Dim_s_Expression prmtpB
dimB xprB)
   = PrimitiveType_Dim_s_Expression (primitiveType prmtpA prmtpB) (dim dimA dimB) (expression xprA
xprB)
expression (Expression_Expression2 xprA1 xprA2)(Expression_Expression2 xprB1 xprB2)
   = Expression_Expression2 (expression xprA1 xprB1)(expression xprA2 xprB2)
expression (Name_Dim_p_Expression nameA dimA xprA)(Name_Dim_p_Expression nameB dimB xprB)
   = Name_Dim_p_Expression (name nameA nameB) (dim dimA dimB) (expression xprA xprB)
expression (Expression_times_or_div_or_mod_Expression1 xprA1 tmphb1A xprA2)
            (Expression_times_or_div_or_mod_Expression1 xprB1 tmphb1B xprB2)
   = Expression_times_or_div_or_mod_Expression1 (expression xprA1 xprB1)(tempHuub1 tmphb1A
tmphb1B)(expression xprA2 xprB2)
expression (Expression_plus_or_minus_Expression1 xprA1 tmphb2A xprA2)
            (Expression_plus_or_minus_Expression1 xprB1 tmphb2B xprB2)
   = Expression_plus_or_minus_Expression1 (expression xprA1 xprB1)(tempHuub2 tmphb2A
tmphb2B)(expression xprA2 xprB2)
expression (Expression_shift_left_or_shift_right_or_Expression1 xprA1 tmphb3A xprA2)
            (Expression_shift_left_or_shift_right_or_Expression1 xprB1 tmphb3B xprB2)
   = Expression_shift_left_or_shift_right_or_Expression1 (expression xprA1 xprB1)(tempHuub3 tmphb3A
tmphb3B)(expression xprA2 xprB2)
expression (Expression_lt_or_gt_or_le_or_ge_Expression1 xprA1 tmphb4A xprA2)
            (Expression_lt_or_gt_or_le_or_ge_Expression1 xprB1 tmphb4B xprB2)
   = Expression_lt_or_gt_or_le_or_ge_Expression1(expression xprA1 xprB1)(tempHuub4 tmphb4A
tmphb4B)(expression xprA2 xprB2)
expression (Instanceof xprA rftpA) (Instanceof xprB rftpB) = Instanceof (expression xprA xprB) (referenceType
rftpA rftpB)
expression (Expression_equal_or_not_equal_Expression1 xprA1 tmphb5A xprA2)
            (Expression_equal_or_not_equal_Expression1 xprB1 tmphb5B xprB2)
```

```
   = Expression_equal_or_not_equal_Expression1 (expression xprA1 xprB1)(tempHuub5 tmphb5A
tmphb5B)(expression xprA2 xprB2)
expression (Address1 xprA1 xprA2)(Address1 xprB1 xprB2) = Address1 (expression xprA1 xprB1) (expression
xprA2 xprB2)
expression (Expression_Expression3 xprA1 xprA2) (Expression_Expression3 xprB1 xprB2)
   = Expression_Expression3 (expression xprA1 xprB1) (expression xprA2 xprB2)
expression (Bar1 xprA1 xprA2) (Bar1 xprB1 xprB2) = Bar1 (expression xprA1 xprB1) (expression xprA2
xprB2)
expression (And1 xprA1 xprA2) (And1 xprB1 xprB2) = And1 (expression xprA1 xprB1) (expression xprA2
xprB2)
expression (Or1 xprA1 xprA2) (Or1 xprB1 xprB2) =  Or1 (expression xprA1 xprB1) (expression xprA2 xprB2)
expression (Colon2 xprA1 xprA2 xprA3) (Colon2 xprB1 xprB2 xprB3)
   = Colon2 (expression xprA1 xprB1) (expression xprA2 xprB2)(expression xprA3 xprB3)
expression (StatementExpression stmtxprA) (StatementExpression stmtxprB)
   = StatementExpression (statementExpression stmtxprA stmtxprB)
expression (Assignment1 asgA) (Assignment1 asgB) = Assignment1 (assignment asgA asgB)
expression  _ _ = Nothing12



primary :: Primary -> Primary -> Primary
primary (PrimaryNoNewArray prnnaA) (PrimaryNoNewArray prnnaB) = PrimaryNoNewArray
(primaryNoNewArray prnnaA prnnaB)
primary (ArrayCreationExpression acxprA) (ArrayCreationExpression acxprB)
   = ArrayCreationExpression (arrayCreationExpression acxprA acxprB)
primary _ _ = Nothing13


dim :: [Dim] -> [Dim] -> [Dim]
dim dimA dimB = matchingDim dimA dimB

matchingDim :: [Dim] -> [Dim] -> [Dim]
matchingDim dimA dimB
            | m > n = dimB
            | otherwise = dimA
                          where
                          m = length dimA
                          n = length dimB


tempHuub1 :: TempHuub1 ->TempHuub1 ->TempHuub1
tempHuub1 tmphbA tmphbB
            | tmphbA == tmphbB = tmphbA
            | otherwise = Nothing14

tempHuub2 :: TempHuub2 ->TempHuub2 ->TempHuub2
tempHuub2 tmphbA tmphbB
            | tmphbA == tmphbB = tmphbA
            | otherwise = Nothing15

tempHuub3 :: TempHuub3 ->TempHuub3 ->TempHuub3
tempHuub3 tmphbA tmphbB
            | tmphbA == tmphbB = tmphbA
            | otherwise = Nothing16

tempHuub4 :: TempHuub4 ->TempHuub4 ->TempHuub4
tempHuub4 tmphbA tmphbB
            | tmphbA == tmphbB = tmphbA
            | otherwise = Nothing17
```

```
tempHuub5 :: TempHuub5 ->TempHuub5 ->TempHuub5
tempHuub5 tmphbA tmphbB
          | tmphbA == tmphbB = tmphbA
          | otherwise = Nothing18


statementExpression :: StatementExpression -> StatementExpression -> StatementExpression
statementExpression (MethodInvocation mthinvA) (MethodInvocation mthinvB) = MethodInvocation
(methodInvocation mthinvA mthinvB)
statementExpression (ClassInstanceCreationExpression clsinsxprA) (ClassInstanceCreationExpression
clsinsxprB)
  = ClassInstanceCreationExpression (classInstanceCreationExpression clsinsxprA clsinsxprB)
statementExpression (Incr2 xprA) (Incr2 xprB) = Incr2 (expression xprA xprB)
statementExpression (Decr2 xprA) (Decr2 xprB) = Decr2 (expression xprA xprB)
statementExpression (Incr3 xprA) (Incr3 xprB) = Incr3 (expression xprA xprB)
statementExpression (Decr3 xprA) (Decr3 xprB) = Decr3 (expression xprA xprB)
statementExpression (Assignment asgA) (Assignment asgB) = Assignment (assignment asgA asgB)
statementExpression _ _ = Nothing19


assignment :: Assignment -> Assignment -> Assignment
assignment (LeftHandSide_AssignmentOperator_Expression lftsA asgopA xprA)
           (LeftHandSide_AssignmentOperator_Expression lftsB asgopB xprB)
  =LeftHandSide_AssignmentOperator_Expression(leftHandSide lftsA lftsB)(assignmentOperator asgopA
asgopB)(expression xprA xprB)


assignmentOperator :: AssignmentOperator -> AssignmentOperator -> AssignmentOperator
assignmentOperator asgopA asgopB
  | asgopA == asgopB = asgopA
  | otherwise = Nothing20


primaryNoNewArray :: PrimaryNoNewArray -> PrimaryNoNewArray -> PrimaryNoNewArray
primaryNoNewArray (Literal lrlA) (Literal lrlB) = Literal (literal lrlA lrlB)
primaryNoNewArray (This0) (This0) = This0
primaryNoNewArray (This1 nameA) (This1 nameB) = This1 (name nameA nameB)
primaryNoNewArray (Expression1 xprA) (Expression1 xprB) = Expression1 (expression xprA xprB)
primaryNoNewArray (ClassInstanceCreationExpression1 clsinsxprA)(ClassInstanceCreationExpression1
clsinsxprB)
  = ClassInstanceCreationExpression1 (classInstanceCreationExpression clsinsxprA clsinsxprB)
primaryNoNewArray (FieldAccess faA) (FieldAccess faB) = FieldAccess (fieldAccess faA faB)
primaryNoNewArray (MethodInvocation1 mthinvA) (MethodInvocation1 mthinvB)= MethodInvocation1
(methodInvocation mthinvA mthinvB)
primaryNoNewArray (ArrayAccess arraA) (ArrayAccess arraB) = ArrayAccess (arrayAccess arraA arraB)
primaryNoNewArray _ _ = Nothing21


arrayCreationExpression :: ArrayCreationExpression -> ArrayCreationExpression -> ArrayCreationExpression
arrayCreationExpression (New prtpA dimxprsA dimA)(New prtpB dimxprsB dimB)
  = New (primitiveType prtpA prtpB) (dimExpressionList dimxprsA dimxprsB) (dim dimA dimB)
arrayCreationExpression (New1 clsintA dimxprsA dimA)(New1 clsintB dimxprsB dimB)
  = New1 (classOrInterfaceType clsintA clsintB) (dimExpressionList dimxprsA dimxprsB) (dim dimA dimB)
arrayCreationExpression (New2 clsintA dimxprinitA arrinitA)(New2 clsintB dimxprinitB arrinitB)
  = New2(classOrInterfaceType clsintA clsintB)(dimExprInitializedList dimxprinitA
dimxprinitB)(arrayInitializer arrinitA arrinitB)
arrayCreationExpression _ _ = Nothing22
```

```
classInstanceCreationExpression :: ClassInstanceCreationExpression->ClassInstanceCreationExpression
                                    ->ClassInstanceCreationExpression
classInstanceCreationExpression (New_comma clstpA xprsA) (New_comma clstpB xprsB)
   = New_comma (classType clstpA clstpB) (expressionList xprsA xprsB)
classInstanceCreationExpression (New_comma1 clstpA xprsA clsbdA)(New_comma1 clstpB xprsB clsbdB)
   = New_comma1 (classType clstpA clstpB) (expressionList xprsA xprsB) (classBody clsbdA clsbdB)
classInstanceCreationExpression (New_comma clstpA xprsA)(New_comma1 clstpB xprsB _)
   = Nothing23 (classType clstpA clstpB) (expressionList xprsA xprsB)
classInstanceCreationExpression (New_comma1 clstpA xprsA _)(New_comma clstpB xprsB)
   = Nothing23 (classType clstpA clstpB) (expressionList xprsA xprsB)


methodInvocation :: MethodInvocation ->MethodInvocation ->MethodInvocation
methodInvocation (NameMethodInvocation nameA xprsA)(NameMethodInvocation nameB xprsB)
   =  NameMethodInvocation (name nameA nameB) (expressionList xprsA xprsB)
methodInvocation (PrimaryMethodInvocation prA idA xprsA)(PrimaryMethodInvocation prB idB xprsB)
                  | idA == idB  = PrimaryMethodInvocation (primary prA prB) idA (expressionList xprsA
xprsB)
                  | otherwise = PrimaryMethodInvocation (primary prA prB) "No match" (expressionList xprsA
xprsB)
methodInvocation (SuperMethodInvocation idA xprsA)(SuperMethodInvocation idB xprsB)
                  | idA == idB = SuperMethodInvocation idA (expressionList xprsA xprsB)
                  | otherwise = SuperMethodInvocation "No match" (expressionList xprsA xprsB)


leftHandSide :: LeftHandSide -> LeftHandSide -> LeftHandSide
leftHandSide (Name2 nameA) (Name2 nameB) = Name2 (name nameA nameB)
leftHandSide (FieldAccess1 faA) (FieldAccess1 faB) = FieldAccess1 (fieldAccess faA faB)
leftHandSide (ArrayAccess1 arraA) (ArrayAccess1 arraB) = ArrayAccess1 (arrayAccess arraA arraB)
leftHandSide _ _ = Nothing24


fieldAccess :: FieldAccess -> FieldAccess -> FieldAccess
fieldAccess (Name1 nameA) (Name1 nameB) = Name1 (name nameA nameB)
fieldAccess (Primary_Identifier prA idA) (Primary_Identifier prB idB)
            | idA == idB = Primary_Identifier (primary prA prB) idA
            | otherwise = Primary_Identifier (primary prA prB) "No match"


arrayAccess :: ArrayAccess ->ArrayAccess ->ArrayAccess
arrayAccess (Name_Expression nameA xprA) (Name_Expression nameB xprB)=Name_Expression (name
nameA nameB) (expression xprA xprB)
arrayAccess (PrimaryNoNewArray_Expression prnnaA xprA)(PrimaryNoNewArray_Expression prnnaB xprB)
   = PrimaryNoNewArray_Expression (primaryNoNewArray prnnaA prnnaB) (expression xprA xprB)
arrayAccess _ _ = Nothing25


dimExpression :: DimExpr -> DimExpr -> DimExpr
dimExpression (Expression2 xprA)(Expression2 xprB) = Expression2 (expression xprA xprB)


arrayInitializer :: ArrayInitializer -> ArrayInitializer -> ArrayInitializer
arrayInitializer (Comma_comma vrinitsA mbprtA)(Comma_comma vrinitsB mbprtB)
   | mbprtA == mbprtB = Comma_comma (variableInitializerList vrinitsA vrinitsB) mbprtA
   | otherwise = Comma_comma (variableInitializerList vrinitsA vrinitsB) Nothing
```

```
dimExprInitialized :: DimExprInitialized ->DimExprInitialized ->DimExprInitialized
dimExprInitialized (Expression_opt mbxprA) (Expression_opt mbxprB)
   |(mbxprA == Nothing) || (mbxprB == Nothing) = Expression_opt Nothing
   |otherwise = Expression_opt (Just (expression mbxprA_ mbxprB_))
                  where
                  (Just mbxprA_) = mbxprA
                  (Just mbxprB_) = mbxprB


literal :: Literal -> Literal -> Literal
literal (IntegerLiteral ilA) (IntegerLiteral ilB) = IntegerLiteral (integerLiteral ilA ilB)
literal (FloatingPointLiteral fplA)(FloatingPointLiteral fplB) = FloatingPointLiteral (floatingPointLiteral fplA
fplB)
literal (BooleanLiteral blA)(BooleanLiteral blB)= BooleanLiteral (booleanLiteral blA blB)
literal (CharacterLiteral clA)(CharacterLiteral clB) = CharacterLiteral (characterLiteral clA clB)
literal (StringLiteral slA)(StringLiteral slB) = StringLiteral (stringLiteral slA slB)
literal (NullLiteral nlA) (NullLiteral nlB) = NullLiteral (nullLiteral nlA nlB)
literal _ _ = Nothing27


integerLiteral :: IntegerLiteral ->IntegerLiteral ->IntegerLiteral
integerLiteral (DecimalIntegerLiteral dilA)(DecimalIntegerLiteral dilB) =
DecimalIntegerLiteral(decimalIntegerLiteral dilA dilB)
integerLiteral (HexIntegerLiteral hxA)(HexIntegerLiteral hxB)= HexIntegerLiteral (hexIntegerLiteral hxA hxB)
integerLiteral (OctalIntegerLiteral oclA)(OctalIntegerLiteral oclB) = OctalIntegerLiteral (octalIntegerLiteral
oclA oclB)
integerLiteral _ _ = Nothing28


octalIntegerLiteral :: OctalIntegerLiteral->OctalIntegerLiteral->OctalIntegerLiteral
octalIntegerLiteral oclA oclB
   | oclA == oclB = oclA
   | otherwise = "No match"


hexIntegerLiteral :: HexIntegerLiteral ->HexIntegerLiteral ->HexIntegerLiteral
hexIntegerLiteral hxA hxB
   | hxA == hxB = hxA
   | otherwise = "No match"


nullLiteral :: NullLiteral ->NullLiteral ->NullLiteral
nullLiteral nlA nlB
   | nlA == nlB = nlA
   | otherwise = "No match"



stringLiteral :: StringLiteral ->StringLiteral ->StringLiteral
stringLiteral slA slB
   | slA == slB =  slA
   | otherwise = "No match"


characterLiteral :: CharacterLiteral ->CharacterLiteral ->CharacterLiteral
characterLiteral clA clB
   | clA == clB = clA
   | otherwise = "No match"
```

```
booleanLiteral :: BooleanLiteral ->BooleanLiteral ->BooleanLiteral
booleanLiteral blA blB
   | blA == blB = blA
   | otherwise = "No match"


floatingPointLiteral :: FloatingPointLiteral ->FloatingPointLiteral ->FloatingPointLiteral
floatingPointLiteral fplA fplB
   | fplA == fplB = fplA
   |  otherwise = "No match"


decimalIntegerLiteral :: DecimalIntegerLiteral ->DecimalIntegerLiteral ->DecimalIntegerLiteral
decimalIntegerLiteral dmlA dmlB
   | dmlA == dmlB = dmlA
   | otherwise = "No match"


methodDeclarator :: MethodDeclarator ->MethodDeclarator ->MethodDeclarator
methodDeclarator (Comma2 idA fpsA) (Comma2 idB fpsB)
   = Comma2 idA (formalParameter fpsA fpsB)
methodDeclarator (MethodDeclarator mdA) (MethodDeclarator mdB) = MethodDeclarator (methodDeclarator
mdA mdB)
methodDeclarator a@(Comma2 _ _)(MethodDeclarator mdB) = methodDeclarator a mdB
methodDeclarator (MethodDeclarator mdA) b@(Comma2 _ _) = methodDeclarator mdA b


methodDeclaration :: MethodDeclaration ->MethodDeclaration ->MethodDeclaration
methodDeclaration (MethodHeader_MethodBody mhA mbA)(MethodHeader_MethodBody mhB mbB)
   = MethodHeader_MethodBody (methodHeader mhA mhB) (methodBody mbA mbB)

methodHeader :: MethodHeader ->MethodHeader ->MethodHeader
methodHeader (Modifier_s_Type_MethodDeclarator_Throws_opt modsA tpA mdA thA)
                    (Modifier_s_Type_MethodDeclarator_Throws_opt modsB tpB mdB thB)
          = Modifier_s_Type_MethodDeclarator_Throws_opt (modifier modsA modsB)(typeMatching tpA tpB)
                                                                    (methodDeclarator mdA mdB) (throws
thA thB)
methodHeader (Void modsA mdA thA)(Void modsB mdB thB)
   = Void (modifier modsA modsB)(methodDeclarator mdA mdB)(throws thA thB)
methodHeader (Void modsA mdA thA)(Modifier_s_Type_MethodDeclarator_Throws_opt modsB tpB mdB
thB)
   = Void (modifier modsA modsB)(methodDeclarator mdA mdB)(throws thA thB)
methodHeader (Modifier_s_Type_MethodDeclarator_Throws_opt modsA tpA mdA thA)(Void modsB mdB
thB)
   = Modifier_s_Type_MethodDeclarator_Throws_opt (modifier modsA modsB) tpA (methodDeclarator mdA
mdB) (throws thA thB)


formalParameter :: [FormalParameter]->[FormalParameter]->[FormalParameter]
formalParameter a b = formalParameterList a b b
formalParameterList :: [FormalParameter]->[FormalParameter]->[FormalParameter]->[FormalParameter]
formalParameterList (_:tl) [] orgB = formalParameterList tl orgB orgB
formalParameterList [] _ _ = []
formalParameterList a@(hdA:tlA) (hdB:tlB) orgB
   | hdA == hdB = [hdA]++formalParameterList tlA orgB orgB
   | otherwise = formalParameterList a tlB orgB
```

```
comparingFormalParameter :: FormalParameter ->FormalParameter ->FormalParameter
comparingFormalParameter (Type_VariableDeclaratorId tpA vdA)(Type_VariableDeclaratorId tpB vdB)
   = Type_VariableDeclaratorId (typeMatching tpA tpB) (variableDeclaratorId vdA vdB)


throws :: Maybe Throws ->Maybe Throws ->Maybe Throws
throws Nothing _ = Nothing
throws _ Nothing = Nothing
throws (Just(Throws_comma clspA))(Just(Throws_comma clspB)) = Just(Throws_comma (classTypeList clspA
clspB))


classTypeList :: [ClassType] ->[ClassType] ->[ClassType]
classTypeList clstpA clstpB = maximumMatching clstpA clstpB


classType :: ClassType ->ClassType ->ClassType
classType (ClassOrInterfaceType1 clsintA)(ClassOrInterfaceType1 clsintB)
   = ClassOrInterfaceType1 (classOrInterfaceType clsintA clsintB)


methodBody :: MethodBody ->MethodBody ->MethodBody
methodBody (Block1 blckA)(Block1 blckB) =Block1 ( block blckA blckB)
methodBody (Semicolon3)(Semicolon3) = Semicolon3
methodBody _ _ = Nothing31


block :: Block -> Block -> Block
block (BlockStatement_s bsA)(BlockStatement_s bsB) = BlockStatement_s (blockStatement bsA bsB)


blockStatement :: [BlockStatement]->[BlockStatement]->[BlockStatement]
blockStatement bsA bsB = maximumMatching bsA bsB


variableInitializerList :: [VariableInitializer] -> [VariableInitializer]->[VariableInitializer]
variableInitializerList visA visB = maximumMatching visA visB


dimExpressionList :: [DimExpr]->[DimExpr]->[DimExpr]
dimExpressionList desA desB = maximumMatching desA desB


dimExprInitializedList :: [DimExprInitialized]->[DimExprInitialized]->[DimExprInitialized]
dimExprInitializedList deiA deiB = maximumMatching deiA deiB
```

File JavaMetrics.hs task is to construct abstract syntax trees.

```
module Main where

import Java
import JavaATermConvertibleInstances
import JavaTermInstances
import JavaChaseImports
import ChaseImports
import StrategyLib
import ATermLib
import System
import DTDJavaMetrics
import DTDJavaMetricsTermInstances
import Text.XML.HaXml.Xml2Haskell
import IO
import Monad
import NameTheme
import List

import MatchingWStr


main =
   do
   args <- getArgs
   cusA <- javaChaseImports [(args!!0)] [(args!!1)]
   cusB <- javaChaseImports [(args!!2)] [(args!!3)]
   infCusA <- getInterfaces cusA
   clsCusA <- getClasses cusA (args!!1)
   sprClsA <- getSuperClasses cusA
   infCusB <- getInterfaces cusB
   clsCusB <- getClasses cusB (args!!1)
   sprClsB <- getSuperClasses cusB
   elClsCusA <- eliminateInheritedMethods clsCusA (infCusA++sprClsA)
   elClsCusB <- eliminateInheritedMethods clsCusB (infCusA++sprClsA)
   trAd <- deleteInheretedMethodsBody elClsCusA
   trBd <- deleteInheretedMethodsBody elClsCusB
    commonTree <- compareCUNew trAd trBd
   commonTree2 <- compareCUNew clsCusA clsCusB
   hPutStr stdout ("£\n"++show trAd)
   hPutStr stdout ("£\n"++show trBd)
   hPutStr stdout ("£\n"++show commonTree)
   hPutStr stdout ("£\n"++show commonTree2)
   hPutStr stdout ("£\n"++show clsCusA)
   hPutStr stdout ("£\n"++show clsCusB)


compareCU :: ( Monad m) => [CompilationUnit] -> m CompilationUnit
compareCU cus = return (compilationUnit tree1 tree2)
                        where
                        tree1 = cus !! 1
                         tree2 = cus !! 2


compareCUNew :: ( Monad m) => [CompilationUnit] ->[CompilationUnit] -> m CompilationUnit
compareCUNew tA tB = return (compilationUnit (head tA) (head tB))
```

```
getInterfaces :: (Term t, Monad m) => t -> m [CompilationUnit]
getInterfaces cus = applyTU (full_tdTU worker) cus
                    where
                    worker = constTU [] `adhocTU` inf
                    inf i = case i of
                                (a@(CU pg im [InterfaceDeclaration id])) -> return [a]
                                _ -> return []


getClasses :: (Term t, Monad m) => t-> String -> m [CompilationUnit]
getClasses cus name= applyTU (full_tdTU worker) cus
   where
   worker = constTU [] `adhocTU` cls
   cls c = case c of
           (a@(CU pg im [ClassTypeDeclaration (Class1 _ id _ _ _)])) -> if name == id then return [a]
                                                                        else return []
           _ -> return []


getMethodParameter :: (Term t) => t -> [FormalParameter]
getMethodParameter cus = head fp
                         where
                         [fp] = getMethParameter cus

getMethParameter :: (Term t, Monad m) => t -> m [[FormalParameter]]
getMethParameter cus = applyTU (full_tdTU worker) cus
                       where
                       worker = constTU [[]] `adhocTU` pm
                       pm p = case p of
                                  (Comma2 _ fpm) -> return [fpm]
                                  _ -> return []


getSuperClasses :: (Term t, Monad m) => t -> m [CompilationUnit]
getSuperClasses cus = applyTU (full_tdTU worker) cus
   where
   worker = constTU [] `adhocTU` spr
   spr s = case s of
           (Class1 _ _ (Just (Extends (ClassOrInterfaceType1 (Name (Identifier_p ids))))) _ _) -> return sprclss
                   where
                   [sprclss] = getSprclass cus (head ids)
           _ -> return []


getSprclass :: (Term t, Monad m) => t ->String -> m [CompilationUnit]
getSprclass cus name = applyTU (full_tdTU worker) cus
   where
   worker = constTU [] `adhocTU` spr
   spr s = case s of
           (a@(CU pg im [ClassTypeDeclaration (Class1 _ id _ _ _)])) -> if id == name then return [a] else
return []
           _ -> return []


eliminateInheritedMethods :: (Term t, Monad m) => t -> [CompilationUnit] -> m t
eliminateInheritedMethods cus interfaces
```

```
   = applyTP (full_tdTP worker) cus
     where
     worker = idTP  `adhocTP` meths
     meths mt = case mt of
                    (a@(Comma2 name parameter)) -> if inheritedMeth then
                                                     return (Comma2 ("InheritedMethod: "++name) parameter)
                                                     else return a
                                                        where
                                                        inheritedMeth     =     checkInheritedMethod     ((show
name)++(show parameter)) interfaces
                    (b@(_)) -> return b



deleteInheretedMethodsBody :: (Term t, Monad m) => t -> m t
deleteInheretedMethodsBody cus = applyTP (full_tdTP worker) cus
   where
   worker = idTP `adhocTP` del
   del d = case d of
             (a@(MethodHeader_MethodBody mh mb)) -> if (chkMeth mh) then
                                                      return (MethodHeader_MethodBody mh Semicolon3)
                                                      else return a
             -- (b@(_)) -> return b



chkMeth :: (Term t)=> t -> Bool
chkMeth cus = isPrefixOf "\"InheritedMethod:" mets
               where
                    mets = getMethodName cus



getNameParameter :: (Term t) => t -> (Identifier, [FormalParameter])
getNameParameter methHeader
   | mp == [] = ("" ,[])
   | otherwise = head mp
                    where
                    [mp] = getIdPara methHeader


getIdPara :: (Term t, Monad m) => t -> m [(Identifier, [FormalParameter])]
getIdPara methHeader = applyTU (full_tdTU worker) methHeader
                         where
                         worker = constTU [] `adhocTU` np
                         np (Comma2 name parameter) = return [(name,parameter)]


checkInheritedMethod :: String -> [CompilationUnit] -> Bool
checkInheritedMethod sign interfaces
   | methodIsInInterface = True
   | otherwise = False
                    where
                    methodIsInInterface = elem sign interfaceSign

                    interfaceSign = lines(head(getMethodSignsFromInterfaces interfaces))


getMethodSignsFromInterfaces :: (Term t) => t ->[String]
```

```
getMethodSignsFromInterfaces cus
   = applyTU (full_tdTU worker) cus
     where
     worker = constTU [] `adhocTU` signs
     signs (Comma2 name parameter) = [((show name)++(show parameter)++"\n")]
```

The file SequenceMatching.hs contains the functions unorderedMaximumMatching and orderedMaximumMatching.

```
module SequenceMatching where

import Array

seqMatching :: (Eq t)=>[t] -> [t] ->  Array (Int, Int) Int
seqMatching seqA seqB = a
   where
   a = array ((0,0),(m,n)) ([((i,0), 0) | i <- [0..m]] ++
                            [((0,j), 0) | j <- [0..n]] ++
                            [((i,j), returnMax (a!(i,j-1)) (a!(i-1,j)) ((a!(i-1,j-1)) + compareValue i seqA j seqB)) | i
                            <- [1..m], j <- [1..n]])
   m = length seqA
   n = length seqB


returnMax :: Int -> Int -> Int -> Int
returnMax a b c
          | a >= b  = max a c
          | otherwise         = max b c


compareValue :: (Eq t)=>Int -> [t] -> Int -> [t] -> Int
compareValue indexA arrayA indexB arrayB
          | lookup indexA arrayA_ == lookup indexB arrayB_          =          1
          | otherwise                                               =          0
         where
            arrayA_ = zip [1, 2..] arrayA
            arrayB_ = zip [1,2..] arrayB


findingSubSeq :: [(Int,Int)] -> [t] -> [t]
findingSubSeq [] _= []
findingSubSeq ((i,_):tail) arrayA = v : findingSubSeq tail arrayA
                    where
                            v = getValue  arrayA_ i
                            arrayA_ = zip [1,2..] arrayA

getValue :: [(Int, t)] -> Int -> t
getValue ((i,v):rest) index
          | i == index = v
          | otherwise = getValue rest index


exSearchNew :: Array (Int, Int) Int -> (Int, Int) -> Int -> (Int, Int) -> [(Int, Int)]
exSearchNew a (i, j) c (m, n)
```

```
        | j > n     = exSearchNew a (i+1, 1) c (m, n)
        | i > m    = []
        | c < newC          = [(i, j)] ++ exSearchNew a (i+1, j+1) newC (m, n)
        | c == newC && i /= m && c < horC          = exSearchNew a (i+1, j) c (m, n)
        | c == newC && j /= n && c < verC          = exSearchNew a (i, j+1) c (m, n)
        | otherwise         = exSearchNew a (i+1, j+1) c (m, n)
        where
                newC = a!(i, j)
                horC = a!(i+1, j)
                verC = a!(i, j+1)


orderedMaximumMatching :: (Eq t)=>[t] -> [t] -> [t]
orderedMaximumMatching seqA seqB
   | (length seqA)== 0 || (length seqB)== 0 = []
   | otherwise = commonSeq
                                where
                                commonSeq = findingSubSeq nodesInPair seqA
                                nodesInPair = exSearchNew multiA (0, 0) 0 (m, n)
                                multiA = seqMatching seqA seqB
                                m = length seqA
                                n = length seqB


unorderedMaximumMatching :: (Eq t) => [t] -> [t] -> [t]
orderedMaximumMatching seqA seqB = auxUMM A B B
auxUMM :: (Eq t) => [t] -> [t] -> [t] -> [t]
auxUMM [] _ _ = []
auxUMM (_:tlA) [] orgB = auxUMM tlA orgB orgB
auxUMM a@(hd:tlA) (hdB:tlB) orgB
        | hdA = = hdB     = [hdA] ++ auxUMM tlA orgB orgB
        | otherwise         = auxUMM a tlB orgB
```