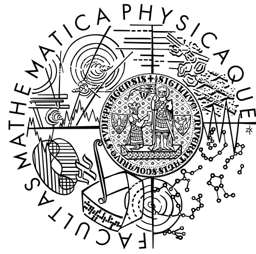

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Jakub Saksa

Syntax-driven duplicate-code detection

Department of Software Engineering

Supervisor of the master thesis: RNDr. David Bednárek, Ph.D.
Study programme: Software and Data Engineering
Study branch: Software Engineering

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor RNDr. David Bednárek, Ph.D., for the patient guidance and advice he has provided throughout my time working on this project.

I would also like to thank my family and friends for their continued support and encouragement during my studies.

Title: Syntax-driven duplicate-code detection

Author: Jakub Saksa

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstract: Duplicate code occurs in source files for different reasons. In many cases the motivation for copying the code is laziness of a programmer, or an attempt to use an alien source code. Over the years, multiple methods for detection of the duplicate source code have been developed. Approaches vary in the ways they analyze the code, focusing on different representations of the program. Methods based on the analysis of the syntactic properties of the source code often use abstract syntax trees. By examining the tree representation instead of the textual representation of the code, these methods are able to detect duplicate code that underwent formatting changes as well as changes to the names of identifiers. Duplicate code fragments are discovered by identifying the subtrees of the same shape. After the suspicious parts of the tree are identified, further examination of AST properties determines to what extent the code was copied. In this work we develop a system for duplicate code detection based on AST comparison.

Keywords: duplicate-code detection, syntax-driven comparison, abstract syntax tree

Contents

Introduction **4**

1 Source code clone detection **5**
1.1 Categories of clones . . . 5
1.2 Clone detection techniques . . . 8
1.2.1 Text based approach . . . 8
1.2.2 Token based approach . . . 8
1.2.3 Tree based approach . . . 8
1.2.4 Graph based approach . . . 9
1.2.5 Metric based approach . . . 9
1.2.6 Hybrid based approach . . . 9
1.3 Clone detection techniques comparison . . . 9

2 Abstract syntax trees and clone detection **11**
2.1 AST introduction . . . 11
2.2 Scenario analysis . . . 12
2.2.1 Format alternation . . . 13
2.2.2 Identifier renaming . . . 13
2.2.3 Statement reordering . . . 14
2.2.4 Control replacement . . . 15
2.2.5 Code insertion . . . 16
2.3 Analyzing clones using ASTs . . . 16
2.3.1 Fingerprinting . . . 19

3 Architecture **21**
3.1 Parser . . . 23
3.2 Hasher . . . 24
3.3 Database layer . . . 25
3.4 Comparer . . . 27

4 PyPlag **28**
4.1 Python . . . 28
4.1.1 Overview . . . 28
4.1.2 Syntax and semantics . . . 29
4.1.3 Typing . . . 31
4.1.4 Uses . . . 31
4.2 Implementation details . . . 32
4.2.1 Parser . . . 33
4.2.2 Hasher . . . 34
4.2.3 Database layer . . . 37
4.2.4 Comparer . . . 40
4.2.5 Frontend . . . 44
4.3 Extending the system . . . 44
4.4 User guide . . . 46
4.4.1 Build . . . 46

4.4.2 PyPlag API . . . 47
4.4.3 PyPlag GUI . . . 48

5 Experiments **52**
5.1 Home assignment . . . 53
5.2 Exam assignment . . . 53
5.3 Handwriting . . . 54

Conclusion **59**

Bibliography **60**

List of Figures **63**

List of Tables **64**

List of Abbreviations **65**

Appendix A Python 3.7.3 grammar **66**

Appendix B Output of astexport **69**

Appendix C Electronic attachments **71**

Introduction

Duplicate source code is frowned upon in software development. One of the first things taught in beginner programming courses is DRY (don't repeat yourself) principle, which tells us that every idea should be written in code only once and reused when needed. By following this rule we make our source code more manageable, less bug prone and easier to understand. This is especially useful for projects with large code bases produced by multiple people. In these projects the DRY principle can be easily violated either by laziness of a programmer or by a programmer not realizing that a certain feature was already implemented by someone else. Cut-and-pasting of the code is often used to save time while writing code. It is faster and more convenient than declaration of new procedure which would hold the duplicate code. While this practice is not ideal from the software engineering point of view, most of the projects in the industry are short-lived, therefore the code maintenance is not the priority. In case the project is successful, duplicate code detection can be done in retrospect.

However, violation of DRY principle is not the sole reason for existence of duplicate code. In academic environment duplicate code can be interpreted as plagiarism, in case the code fragment is present in source files created by different authors working on the same or similar task. Scenarios for this would be e.g. a homework assignment, an exam assignment, or even a thesis. Discovering plagiarism is often impossible without help of automation. Multiple systems were developed over the years because of this e.g. MOSS [1], CCFinder [2] or GPLAG [3]. Survey of commonly used techniques can be found in section 1.3.

This work deals with duplicate code fragment detection based on AST comparison of source files. AST (abstract syntax tree) is a representation of code, mostly used in compiler frontends as the result of the syntax analysis phase. The fact that this structure precisely captures all relevant properties of the source code makes it a good base for duplicate detection. Compared to other means of duplicate code detection, it provides a good ration between the complexity of the solution and the complexity of source code duplicates it is able to detect. AST based approach solves the issues of text based methods associated with the changes in the formatting of the code and identifier renaming.

The system we have developed consists of multiple components. Each of these components is assigned with one action that is necessary for duplicate source code detection. Performed actions cover everything from parsing of source code files to forming an output of the system. Components are connected into a pipeline, so output of one component is used as an input for the other part of the system. Pipeline is designed in such a way, that allows for future extension of the system by adding the support for multiple programming languages. Of course each language has specific structure defined by the language grammar. This introduces the need for the pipeline components to be split into two groups - language dependent and language independent components.

The parts of the pipeline handling the parsing of the source code and AST analysis need to be aware of language specifics and therefore are language dependent. Pipeline components responsible for these actions are the parser and the syntactic analyzer. When introducing the processing of a new programming

language into the pipeline, these two components need to be reimplemented.

After the analysis of the source code is finished, its results enter the language independet part of the pipeline. The job of these components is to store the information received from the syntactic analyzer. Later this information is retrieved from the storage and used to determine if duplicate code occurs in the source files. This part of the system is split into two parts - the permanent storage component and the comparer component. Comparer component performs analysis of gathered AST data and the result of this part of the system represents the result of the entire computation. The language dependent part of the pipeline prepares the data and shields the language independent components from the specifics of the language. This makes permanent storage and comparer component reusable for any programming language.

As part of this work we have created an example implementation of the pipeline called PyPlag. Our implementation covers all the aforementioned system components and the language dependent part is able to process source code written in Python. We have also decided to create a frontend in form of a graphical user interface to represent the raw results of the PyPlag pipeline.

The work is divided into multiple parts. Each part has its own chapter. In the first chapter we present a comprehensive overview of duplicate source code detection scenarios, together with a list of commonly used techniques to discover the code clones. After that we discuss the role of ASTs in the context of clone detection in chapter 2. In this part of the work we analyze the ways code duplication scenarios are reflected in the structure of an AST. The discussion of the AST based techniques concludes this part of the work. Chapters 3 and 4 are dedicated to our clone detection system. In the third chapter we present the architecture of the pipeline together with the discussion of the key decisions that influenced the design. Chapter 4 is focused on the PyPlag implementation and covers used algorithms. The user guide and the guide for extending the system are also part of chapter 4. In the last, fifth, chapter we present the results of the experiments.

1. Source code clone detection

To detect duplicate code, we first need to understand what it looks like. If code duplication is the result of cut-and-paste action of a programmer, we will get multiple instances of the same code fragment scattered throughout one or multiple source files. These pieces of code are *clones* of the original code that was copied. In case the copied code is a result of plagiarism, programmer might attempt to alter the code to hide the fact that it is a copy. The code is still a clone of the original because it represents the same idea and solves the same problem, even though it looks different in the text form.

1.1 Categories of clones

To better understand the term *source code clone* we need to define different categories of clones. According to Chen [3] these are the most common cloning strategies.

Format alternation: Original code is altered by adding whitespaces and line breaks where possible. Comments and other elements which do not change meaning of the code are considered format alternation as well.

Original code

```
#include <iostream>
using namespace std;

int main()
{
   const size_t count = 10;

   for(size_t i = 0; i < count;
       ++i)
   {
      cout << i << endl;
   }

   return 0;
}
```

Format alteration

```
#include <iostream>
using namespace std;

int main()
{
   // ugly code
   const size_t count = 10; for(
                        size_t i = 0;
                        i < count;
                        ++i
                    ) { cout << i
                         << endl; }
                         return 0;

}
```

Figure 1.1: Format alternation

Identifier renaming: Names of identifiers can be changed without violating program correctness. Targets of this technique are usually variables and user defined types (classes, structs, etc.). For example, in figure 1.2 there is one constant named *count* that was renamed to *limit* and one variable controlling the *for* loop named *i*, renamed to *j*.

Statement reordering: In some cases, it is possible to change the order of

Original code

```
#include <iostream>
using namespace std;

int main()
{
   const size_t count = 10;

   for(size_t i = 0; i < count;
      ++i)
   {
      cout << i << endl;
   }

   return 0;
}
```

Identifier renaming

```
#include <iostream>
using namespace std;

int main()
{
   const size_t limit = 10;

   for(size_t j = 0; j < limit;
      ++j)
   {
      cout << j << endl;
   }

   return 0;
}
```

Figure 1.2: Identifier renaming

statements in source code without changing the meaning of the program. Changes to the statement order are usually very subtle, because the order forms a part of the program semantics. Bigger changes are almost guaranteed to change the semantics.

Original code

```
#include <iostream>
using namespace std;

int main()
{
   string hello = "Hello";
   cout << hello << " ";
   string world = "World"
   cout << world;
   string exclamation = "!';
   cout << exclamation;

   return 0;
}
```

Statement reordering

```
#include <iostream>
using namespace std;

int main()
{
   string hello = "Hello";
   string world = "World";
   string exclamation = "!';
   cout << hello << " ";
   cout << world;
   cout << exclamation;

   return 0;
}
```

Figure 1.3: Statement reordering

Control replacement: In languages supporting multiple mechanisms for flow control, it is possible to replace one mechanism with another. For example *for* loop can be replaced by *while* loop (figure 1.4). Another example is replacing if(condition){A}else{B} construct by if(condition){B}else{A}, where negation of *condition* is used and bodies of *if* and *else* branches are swapped.

Original code

```
#include <iostream>
using namespace std;

int main()
{
   const size_t limit = 10;

   for(size_t i = 0; i < limit;
       ++i)
   {
      cout << i << endl;
   }

   return 0;

}
```

Control replacement

```
#include <iostream>
using namespace std;

int main()
{
   const size_t limit = 10;
   size_t i = 0;

   while(i < limit)
   {
      cout << i << endl;
      ++i;
   }

   return 0;
}
```

Figure 1.4: Control replacement

Code insertion: To change the structure of the code, immaterial code can be inserted between the statements in such a way that meaning of the code stays the same. In figure 1.5 *for* loop is inserted between the original statements.

Original code

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello ";
   cout << "World!"

   return 0;

}
```

Control insertion

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello ";
   for(size_t i=2; i<1; ++i) {}
   cout << "World!"

   return 0;
}
```

Figure 1.5: Control insertion

These categories cover majority of code clones. However, there are few techniques that do not fall into any of the categories. Examples of such methods are:

- **Changing the type of a numeric value**: It is possible to disguise the source code clone by changing the types of numeric values in places where it does not change the correctness of the program, e.g. changing integer values to floating point values.

- **Splitting the code into subroutines**: Breaking up big blocks of code into multiple methods and then using statements of calling these methods to achieve the original functionality.
- **Using shorthand operators**: In languages supporting shorthand operators to provide quality of life enhancement to programmers, certain statements might be rewritten using these operators. Example is operator += which is just a short version of classic increment statement.

1.2 Clone detection techniques

Clone detection techniques are clasiffied based on how they approach the analysis of source code [4]. Approaches vary from simple text based methods to complex graph analysis.

1.2.1 Text based approach

Text based - also called string based - techniques look at the source code fragments as subsequences of a text. Two fragments are compared textually using multiple text based transformations, e.g. removing white spaces and comments from the code. The goal of the comparison is to identify sequences of identical strings. These are then considered clones. Most of the tools using text based approach are not able to consistently identify clones created by identifier renaming [4]. Nature of the approach also indicates that cloning strategies based on code structure changes are a problem.

Example of text based technique is language independent approach using several transformations on source code proposed by Ducasse et al. [5]. Di Lucca et al. [6] explored the use of Levenshtein distance of source codes.

1.2.2 Token based approach

Token based approach is also called lexical approach. The source code is split into tokens. This is performed by lexical analysis. Tokens form a token sequence which is scanned for duplicates. The fact that tokens are produced by lexical analysis - which uses lexical rules of a language - makes this approach language dependent.

CCFinder [2]uses token based approach to find code clones in C, C++ and Java source files.

1.2.3 Tree based approach

Source code is represented in a form of an abstract syntax tree (AST). AST is a representation of abstract syntactic structure of the code. More information on ASTs can be found in chapter 2. After obtaining AST of the program, the idea is to compare it with another AST to identify identical or similar nodes and subtrees. These structures represent code clones in original source files. Comparison of ASTs can be done with the help of the tree edit distance algorithm [7]. However, the complexity of the algorithm is a problem. With complexity $O(max(m, n)^3)$ for

two trees with m and n nodes, scalability becomes an issue in cases where a large number of trees need to be compared. Baxter et al. [8] deals with this problem by categorizing the trees and then comparing only the trees in the same category. Categorization is performed by hashing a tree. By using locality-sensitive hashing it is possible for near exact trees to fall in same category.

Different approach was explored by Wahler et al. [9]. They converted the AST into XML and subsequently used Frequent Itemset data mining technique to extract the clones.

1.2.4 Graph based approach

This approach uses program dependecy graph (PDG) [10], capturing data flow and control flow of the program, to detect clones both semantically and syntactically. Clones are found by finding subgraph isomorphisms in PDGs. GPLAG [3] is an example of a tool using graph based approach. Using PDGs for clone detection was proved to be the most robust technique [3]. Disadvantage of this approach is the complexity of PDG creation and PDG comparison.

1.2.5 Metric based approach

Instead of analysing the source code directly, the metric based techniques collect metrics about the code. These metrics might be e.g. number of lines, number of methods, classes etc. These are then compared with values collected from other files.

Mayrand et al. [11] presented a metrics based tool which gathers metrics from expressions, functions, layouts (e.g. number of non-empty lines) and control flow. Code fragments with similar metrics are then returned as clones.

1.2.6 Hybrid based approach

The hybrid approach is collection of several aforementioned techniques.

Tree and token based hybrid approach using AST serialization and suffix trees was proposed by Koschke et al. [12]. Suffix tree comparison algorithm is able to identify exact clones. The same approach is used by Microsoft's Phoenix framework [13] for detection of clones at function level.

By combining tree and metric based approaches Jiang et al. [14] developed the method for computing code fragment similarity. Method analyses AST of the program as an object in Euclidean space, combined with local sensitive hashing [15].

1.3 Clone detection techniques comparison

Each method has its pros and cons. Usually there is a tradeoff between complexity of the method and number of scenarios in which it is able do identify clones.

Simple method like text based approach is robust to format alternation due to transformations of source code, namely discarding blank lines and comments, but it is fragile to identifier renaming.

Approaches based on AST are more complex than text based methods. They are able to handle both format alternation and identifier renaming, but as consequence of ignoring data flows, they have difficulties handling statement reordering and control replacement.

Token based approach is robust to identifier renaming because variables of the same type are mapped into the same token. By relying on analysing the sequence of tokens it is sensitive to statement reordering and code insertion. The reason for this is the fact that both of these scenarios change the token sequence. However, most commonly used tools for plagiarism detection *MOSS* [1] and *JPlag* [16], both use token based approach. *MOSS* overcomes fragility to token sequence altering strategies by fingerprinting [17].

Graph based approach is the most complex and as a consequence of that, it is able to discover clones in all aforementioned scenarios. Fragility to code insertion and statement reordering of other methods is removed by taking data and control flow to an account.

Comprehensive comparison of different approaches and their robustness with respect to code cloning scenarios is presented in table 1.1. Term *partial* in the token based approach column is used because some inadequacies of the approach can be partially remedied with use of fingerprinting [18].

	Text	AST	Token	Graph
Format alternation	Yes	Yes	Yes	Yes
Identifier renaming	No	Yes	Yes	Yes
Statement reordering	No	No	Partial	Yes
Control replacement	No	No	Partial	Yes
Code insertion	No	No	No	Yes

Table 1.1: Approach robustness comparison [3]

2. Abstract syntax trees and clone detection

In this chapter we take a closer look at AST as the base for source code clone detection. This includes analysis of how different types of clones influence the shape of an AST and discussion of AST clone detection techiques.

2.1 AST introduction

Abstract syntax tree is a tree representation of the abstract syntactic structure of the source code written in a programming language. Nodes of the tree represent constructs of the language. Abstract syntax does not capture all the details of real syntax, only structure related information. For example, a syntactic construct like a *while(condition){body}* may be denoted by a single node with two children branches. Other kind of tree representation of source code is *concrete syntax tree*, also known as a *parse tree*. The difference is the level of the detail both structures capture. Concrete syntax tree captures all the symbols from real syntax of the language. i.e. parentheses, semicolons, etc.

ASTs are data structures most commonly used in compilers to represent the structure of the code. An AST is usually the product of the syntax analysis phase of compilation. It serves as a representation of the program through subsequent stages that compiler performs. The AST is intensively used during semantic analysis. This is the phase where compiler applies the rules of the language to verify correctness of the code and to convert the code to an intermediate representation.

There are multiple resons why an AST is ideal structure to store the program structure during the process of compilation:

- It can be enhanced with information, either via properties or annotations, for every node.
- AST, compared to the source code, does not contain elements that hold inessential information, like punctuation and delimeters.
- Due to other stages of analysis performed by the compiler, AST can hold extra information about the source code. For example, it may store positions of elements in the source code.

The design of an AST is closely linked with the design of the compiler. Different compilers may present different features, but the core requirements on AST are usually:

- The order of declarations in source code must be preserved.
- The order of statements and other elements like function arguments and operands must be preserved.
- Identifiers and values assigned to them must be stored.

These requirements can be used as guidelines for designing data structures for the AST.

AST should be flexible enough to be able to add an unknown number of children of different types to a node. Requirement for adding unknown number of children nodes comes from the fact that language grammar often allows arbitrary number of nodes for some roles, e.g. statements in the body of a function. The need for multiple types of children nodes to fill the same role comes from grammar rules that define a subset of node types for each role. For example, in the body of a function, there might be multiple types of statements, like assignment statements, conditional statements, loops, etc. All concrete statements are subtypes of an abstract statement. This inheritance replaces grammar rules but is less exact. Constructs that are forbiden by the language grammar may occur in the tree, but in the context of clone detection it does not matter, as we are able to process the AST anyway.

It should also be possible to reconstruct the source code from an AST. Reconstructed source code should be sufficiently similar to the original source code of the program. This feature is useful for visualisation of source code clones. A tree representation of the code is less intuitive than its text form. Reconstructed code does not need to look exactly the same as the original, but it is important that its meaning stays the same.

Due to the complexity of the requirements for an AST it is crucial to follow software development principles. Namely usage of design patterns which enhances modularity and sustainability of the code. Because AST is a tree structure, the most common action performed on it is traversal combined with different operations on the tree nodes. This leads to the use of the *visitor pattern*. Each node type is represented as a respective class. This approach is natural because it follows the structure of grammar rules as closely as possible. For example dividing nodes into multiple categories, such as statement nodes and expression nodes. Concrete node types may differ in their properties and the information they carry but at the same time they have overlaping properties, like information about line numbers. Using inheritance makes it easy to capture this relationship between node types.

2.2 Scenario analysis

AST is just another view of the source code. This means we can analyse this view further instead of analysing the textual representation of the code. The fact that an AST captures all the important information about the nature of the source code implies that changes made to the code during the cloning process are reflected in the structure of the AST. This holds for almost all code clones, except those created via format alternation. In this section we will take a look at each cloning scenario and the way it influences an AST. Visualisation of ASTs was created by Python AST Visualizer tool [19].

2.2.1 Format alternation

Inserting white spaces or comments into the code does not change AST at all. White spaces do not hold any information[1], so there is no point to store them in an AST. Comments, on the other hand, do provide some information, but this information is for the reader of the source code - to provide an explanation of the purpose of the code. It does not change the functionality of the program so AST does not capture it. However, it is possible to store the comments via annotations of nodes in the AST.

In conclusion, format alternation produces source code that translates into the same AST as the original source code. This is ilustrated by figure 2.1. Finding clones in AST pairs created by this scenario is straightforward, all we need to do is to identify subtrees with the exactly same structure.

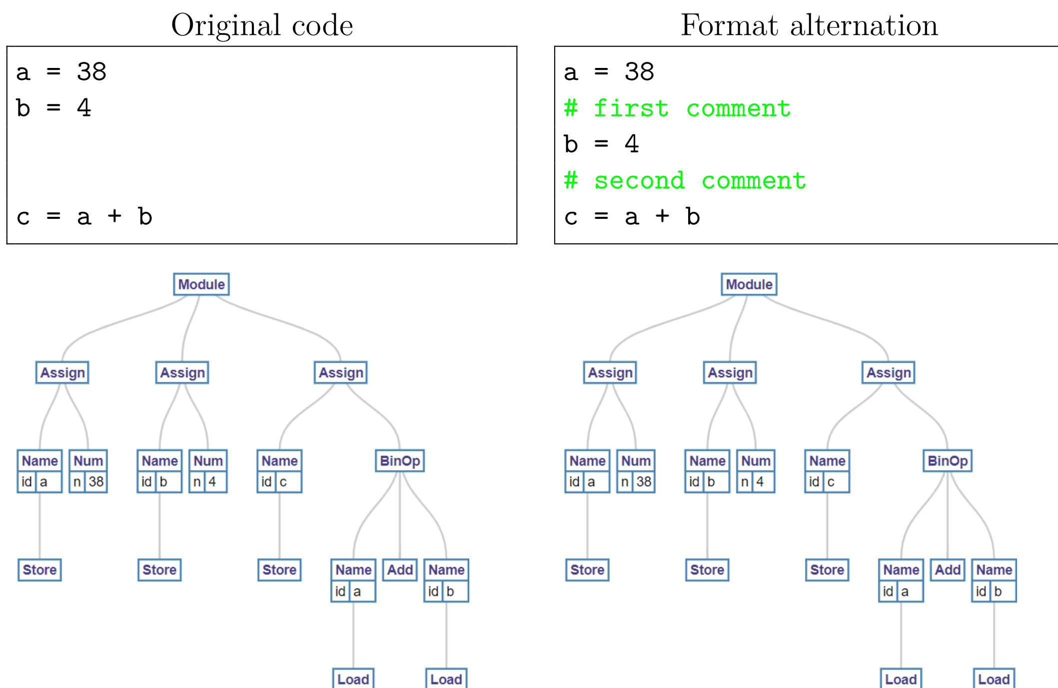

Figure 2.1: AST format alternation

2.2.2 Identifier renaming

Similarly to format alternation, renaming identifiers does not change the shape of an AST. Difference here is that although the type of nodes stays the same, they still denote identifiers, and the properties of nodes change. (figure 2.2). This reflects the change of identifier names in the source code.

To find clones created by identifier renaming we cannot look for exact match of the subtrees. Difference in identifier names would cause two otherwise identical subtrees to be proclaimed different. But if we look at the type of the nodes and further analyse the names of identifier nodes, we get the desired result of

[1]We are talking about source code conforming to all the rules of the language. In some languages, e.g. Python, indentation by white spaces is needed to declare group of statements forming a body of control constructs like for and while loops.

identifying subtrees as clones. Name analysis consists of renaming the identifiers in both trees, so we can verify that they have the same meaning. Algorithm for renaming is described in section 4.2.4. In figure 2.2 we can rename indentifiers a and x to x_1, b and y to x_2, c and z to x_3. After renaming the two pieces of code are identical, so they must have the same meaning. This approach can be applied also in other situations, where it is more important to identify the language construct and not the exact value that was used in the source code.

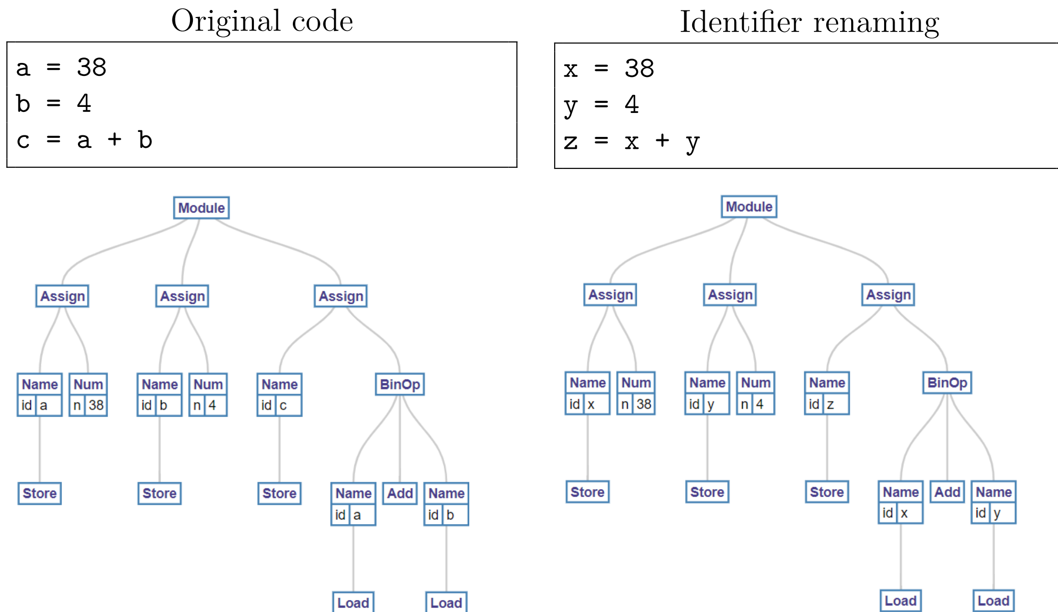

Figure 2.2: AST identifier renaming

2.2.3 Statement reordering

By reordering the executable statements in the source code of the program in such a way that does not change the correctness (figure 2.3), we actually change the shape of the AST representing the program structure. Subtrees representing the statements change their positions in the tree. To be more exact, they swap positions with their siblings. This means that the rest of the tree stays the same and only the subtree containing the statements in rearranged order changes its shape.

However, even this minor change to the structure of the tree requires more complicated approach to find the clones. Comparing entire trees node by node, like in previous two scenarios would fail beacause of their different shapes. Instead, we need to use more sophisticated method. We could take every subtree from the original AST and compare it to every subtree from altered AST to find identical subtrees. To solve this problem we can take advantage of the fact that TED algorithm must solve this subproblem to find the edit distance [7]. During the computation the algorithm creates a mapping between two trees to determine which subtree matches the best with which subtree. As we mentioned in chapter 1, this takes $O(max(m, n)^3)$ time for trees with m and n nodes.

Another approach would be to compute hash of each statement node and then use string edit distance to find a difference between hash values. Time complexity

of this approach is $O(mn)$. Disadvantage of hashing is that a change in a subtree representing an arbitrary statement in the statement sequence is treated as if the statement was completely replaced by another statement. However, the change in the statement might be semantically as significant as replacing the statement.

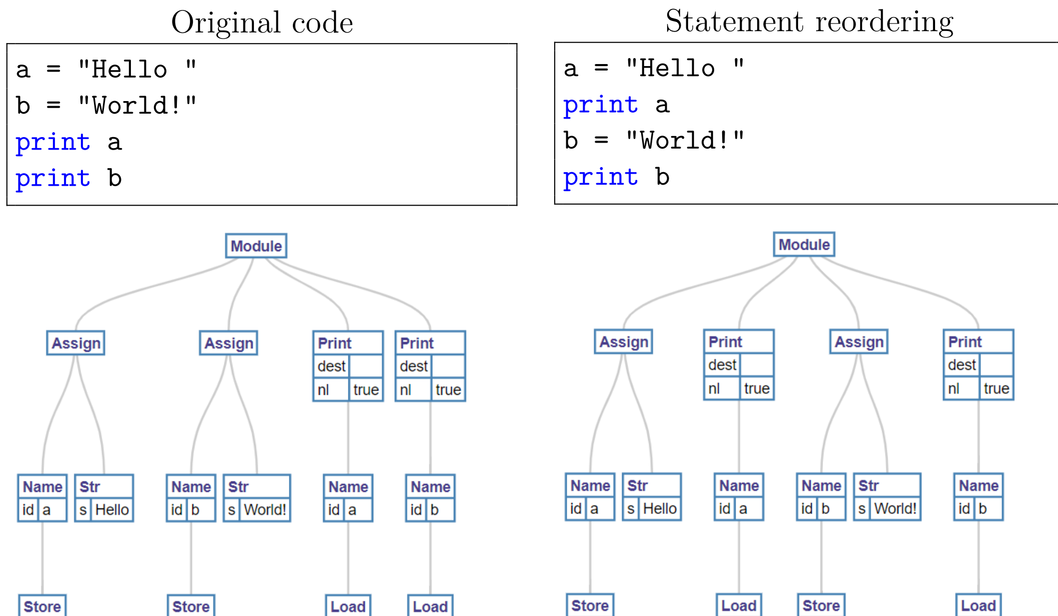

Figure 2.3: AST statement reordering

2.2.4 Control replacement

Another scenario that changes the shape of an AST is control replacement. The difference between this scenario and statement reordering is in the way it changes the AST structure. While changing the order of statements only changes the order of respective subtrees in the AST, control replacement replaces the whole subtree by a new subtree. Even though some of the nodes in the new subtree stay the same, there are also new nodes, reflecting the changes in the source code, e.g. new control flow construct and helper statements which are needed to achieve the same behaviour as in the original source code.

Figure 2.4 is an example of the replacement of *for* loop by *while* loop. Original AST changed noticeably. Instead of root having one child, after control replacement it has two child nodes. This is a consequence of declaring the control variable used by *while* loop. Branch representing control flow construct of the language has the same number of subtrees, but they are not the same. In the original tree, first two branches represent the code controlling the number of iterations of the loop and the third branch represents the work done inside of the loop. In the modified tree, the first and the third branch are controlling the number of iterations, while the second branch represents the work. Clones we want to find are in this case the third branch from the original control subtree and the second branch from the modified control subtree.

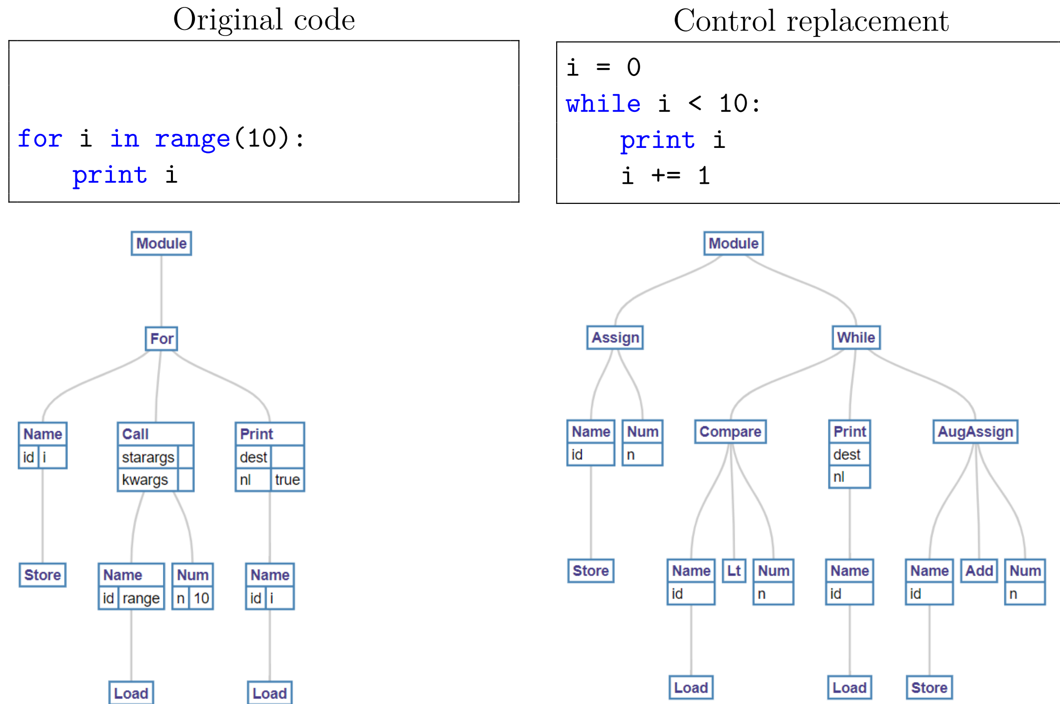

Figure 2.4: AST control replacement

2.2.5 Code insertion

By inserting immaterial code into the original source code of the program, we achieve another modification of an AST. This time no subtree is replaced, but instead a new subtree is created and mixed in between the original child nodes representing the executable statements. Inserted code can be arbitrarily complex as long as it does not influence the correct behavior of the program. Figure 2.5 shows an example of code insertion scenario. Modified AST looks much more complex and bigger[2] than the original AST. In reality, however, we can see that it is the same tree except there is a subtree representing immaterial code inserted between two original branches.

Identifiying the duplicate branches can be done the same way as in previous scenarios in which the shape of an AST was altered.

2.3 Analyzing clones using ASTs

Problem of finding clones in an AST boils down to finding places in ASTs, in which the structural change has occured. There are multiple approaches to this problem used in practice.

Tree edit distance

Because ASTs are tree structures, a natural way of comparing them would be TED algorithm. TED between ordered labeled trees is the minimal-cost sequence of

[2]With respect to the number of nodes in the tree.

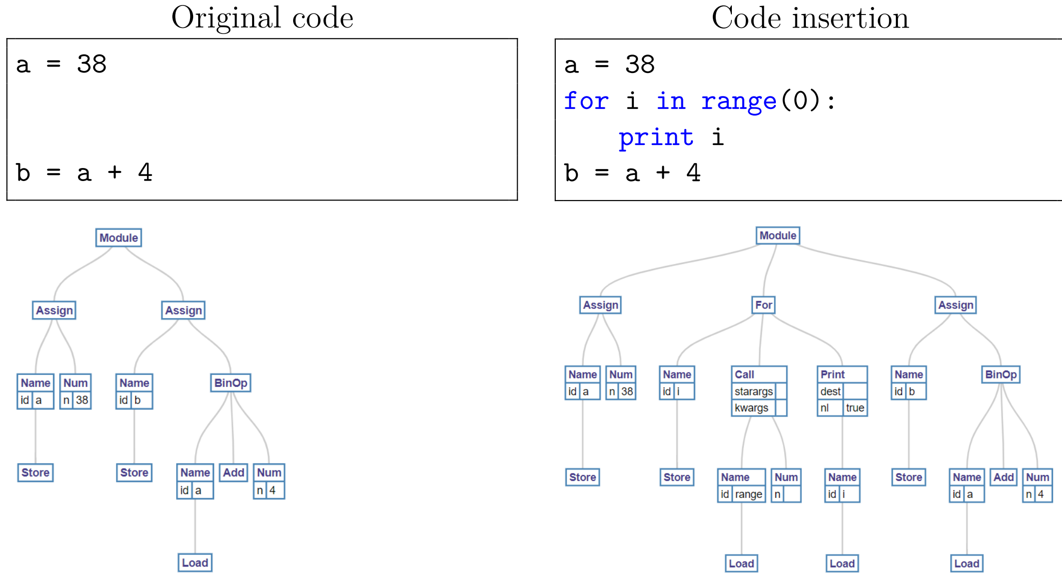

Figure 2.5: AST code insertion

edit operations that transforms a tree into another tree [7]. Tree modifications that are considered an edit operation are:

- **Delete** a node. If this node was not a leaf, its children are connected to the parent of the node.
- **Insert** a node as a child of already existing node.
- **Rename** a node.

A cost might be assigned to each of these operations to fine-tune the process of tree comparison.

The result of the computation is the sequence of edit operations with minimal cost. In our case, we would only take into consideration the cost, which indicates to what extent the trees are similar. Even though we discard significant part of the algorithm output, the remaining part still provides more information than we need. We only need to determine if two trees have identical shape. TED also tells us how much the shapes vary if the trees are not identical. This makes TED a precise candidate for a potential approach, but it is an overkill with respect to the amount of the work that needs to be done to get the result.

Tree kernel

Tree kernel algorithm is used for comparison of tree structures and for measuring their similarity in natural language processing. Between two rooted trees T_1 and T_2 a kernel $K(T_1, T_2)$ can be expressed as an inner product of two vectors:

$$K(T_1, T_2) = \mathbf{h}(T_1) \cdot \mathbf{h}(T_2).$$

Each subtree that is a part of the data set is reffered to with index in range 1,...,n. Based on this indexing, the vector $\mathbf{h}(T)$ is defined as:

$$\mathbf{h}(T) = (h_1(T), h_2(T), ..., h_n(T)).$$

Value h_i denotes the number of occurences of the ith subtree in T [20].

Fu et al. [21] combined kernel algorithm with calculating the weights of nodes in ASTs and TF-IDF retrieval[3]. Weights are calculated for every subtree. Expressions appearing often in one program, but not often in other source codes are assigned greater weight than expressions that appear in every program. After determining the weights, the kernel algorithm is used to detect plagiarism.

To deal with renaming of identifiers and other code cloning techniques, authors of this approach used various tree transformations. For example, all variable tokens are represented by one unified token. Same goes for array indices. Another AST modification is replacing the subtrees representing the expressions with their non-AST representation. Reasoning behind this is that modification of expression as plagiarism technique is not easy to pull off. Therefore expression nodes are represened as in-order traversal strings.

Results of experiments suggest that this approach works well. Compared to the *JPlag* tool it was able to identify up to 3 times as many duplicates and with higher precission [21]. Experiments were carried out on two datasets consisting of student submissions for final exam of the programming course.

Authors also introduced a simpler method, which does not consider the weight of the subtrees. In this case all the subtrees have the same weight of 1.

Common disadvantage of aforementioned approaches is the complexity of the tree comparison. This might become a problem when dealing with the large amount of source files. The problem could be overcomed by preprocessing of ASTs and indexing the preprocessed data. Then the trees would not be compared from scratch every time the computation begins, but instead the comparison would look at previously processed data. However, in case of both approaches there is no clear way of how to efficiently index subresults of the computation.

If we take into consideration how source code clones come into existence, we can see that a complex approach to AST comparison is not needed. In case of plagiarism, we can assume that the plagiator does not understand the code and therefore they are only making simple changes, like renaming identifiers, etc. These changes do not change the structure of the code. Even if the plagiator changes the structure of the code in some places, it cannot be a significant change because the code still needs to function in the same way as the original. If code clones are result of a laziness of the programmer who copy-pastes their own code, we can assume that the structure of the code stays the same. Only change in this case would be in places which can be parametrized, like identifier names and constants.

In this context, we do not need to use a complex algorithm which would look for structural changes in a tree globally. Instead we can assume that a change in the structure of the tree translates into a semantic change in the code. The parts of the trees that are identical, we consider a clone. Identity is in this case an identity of the tree shape.

These assumptions lead to the use of a simpler technique called *fingerprinting*. This approach is able to efficiently detect trees with identical shape.

[3]Term frequency and Inverse document frequency retrieval.

2.3.1 Fingerprinting

Each subtree from an AST is represented with a *fingerprint*. Fingerprint consists of a tuple containing subtree weight[4], its hash value[5] and a pointer to its parent in the tree [22]. After the creation of fingerprints, they are stored in permanent storage, so they can be used when checking for clones in a specific source file. Chilowicz et al. [22] decided to use B+-k-tree, beacause of the ability of the structure to iterate over fingerprints and retrieve the clones with the highest weight. Retrieval of clones from database is done by comparing the fingerprint of input subtree with fingerprints in the database. If the records share the same weight and hash value they are considered identical.

For this approach to be effective, computation of the hash value needs to be fast. Since we want to capture the structure of an AST, the hash function must take into an account every node of the tree. Hash value can be computed in linear time using incremental hash function. This hash function only needs to know hashes of immediate children of the node to create the hash. Acording to the authors of [22] these methods are good candidates for fingerprinting:

- **Sum hashing** counts the number of nodes of the same type in a subtree. The subtree is then represented by a vector consisting of the count for each node type. Hash value is computed as the hash of the vector. This approach discards the structural information about the tree and therefore false positives occur more often.

- **Dyck word hashing** represents each subtree as a Dyck word. Dyck word is serialization of the subtree constructed from tree traversal. The Dyck word for the node n is defined as concatenation:

  $$V_b(n) \cdot D(c_1) \cdots D(c_n) \cdot V_e(n),$$

  where values $D(c_1)$,...,$D(c_n)$ are Dyck words of children nodes of n. Values $V_b(n)$ and $V_e(n)$ mark the beginning and the end of the word, they are computed from node type vector of the node. The word is hashed with Karp-Robin hash function, which can be incrementaly computed [23].

- **Cryptographic hashing** uses cryptographic hash function, like MD5 [24], to hash the tree. This method is completely incremental, because the tree is hashed bottom up, starting from leaves. Leaf hash values are concatenated and form hashes of inner nodes. The whole tree is processed in linear time with the respect to number of nodes in the tree. Hash function using the cryptographic hash function C can be defined as:

  $$H_C(t) = C(V(r) \cdot H_C(t_1) \cdot H_C(t_2) \cdots H_C(t_n)).$$

  In the formula t represents the tree the hash value is computed for and t_1,...,t_n are child nodes of the root r. Concatenation of child node hashes is prefixed with the information about the root node. In case that hash values produced by the hash function are too big, they might be shortened by selecting fixed number of the first or the last bytes.

[4] Size of the subtree.

[5] Capturing structural properties of a subtree.

Hashing methods migh be altered, so that they do not reflect some properties of an AST. For example the same hash value might be used for nodes which represent different loop constructs. This can help to detect clones created by control replacement. Same logic can be applied in case of primitive types, e.g. all numeric types can be grouped together and the whole group is represented by the same hash value.

The approach we have decided to implement as a part of this work is based on fingerprinting. However, we have modified some parts of the technique so it better suits our needs. More information about the clone detection process is provided in chapter 4.

3. Architecture

The goal of this work was to develop a system for duplicate source code detection. This system should be able to handle one-to-many comparison of source files. With this goal in mind we have designed the system as a pipeline consisting of these components:

- parser,
- hasher,
- database layer,
- comparer.

In addition to these components, the system provides an API with basic functionality needed to create a *visualizer* component used as frontend for representation of the results computed by the pipeline.

The workflow of the system depends on the action user decided to carry out. There are two actions that are needed to perform the source code clone detection: **building** the database and **comparing** a source file to the files in the database.

While building the database, there is no need for actual comparison of source code, therefore the *comparer* component is left out during this action. The process is captured by figure 3.1. The source files are processed one by one. First step is to parse the input source file. Output of the *parser* is an AST representing the source code. The AST is subsequently processed by the *hasher* to prepare the tree for permanent storage. The permanent storage is represented by the *database layer*, where additional actions necessary for storing the trees are performed.

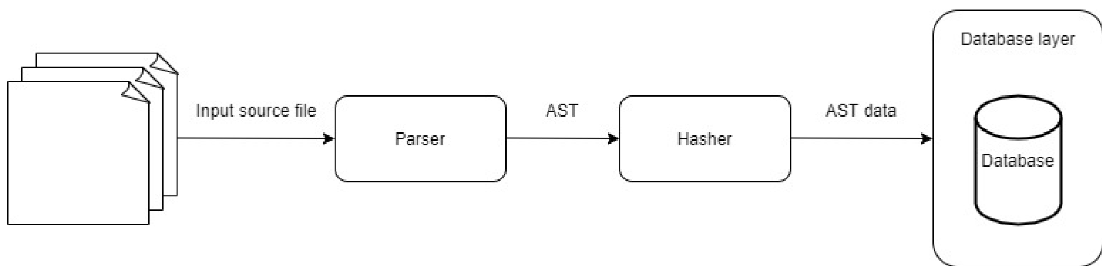

Figure 3.1: Building the database

All components of the system are involved when it comes to the actual detection of the source code clones. As mentioned before, detection of clones is done by comparing the AST representation of input source file to the AST data stored in the database. To extract the AST from the source code and prepare it for comparison, the *parser* and the *hasher* are used again, in the same way as in the previous use case. The difference is that after processing the AST the output of the *hasher* component is not stored in the database, but it is forwarded to the *comparer*. *Comparer* identifies source code clones[1] based on the input AST data

[1] If there are any.

and the database AST data. List of identified clones forms the output of the *comparer* and the output of the whole action. This process is captured by figure 3.2.

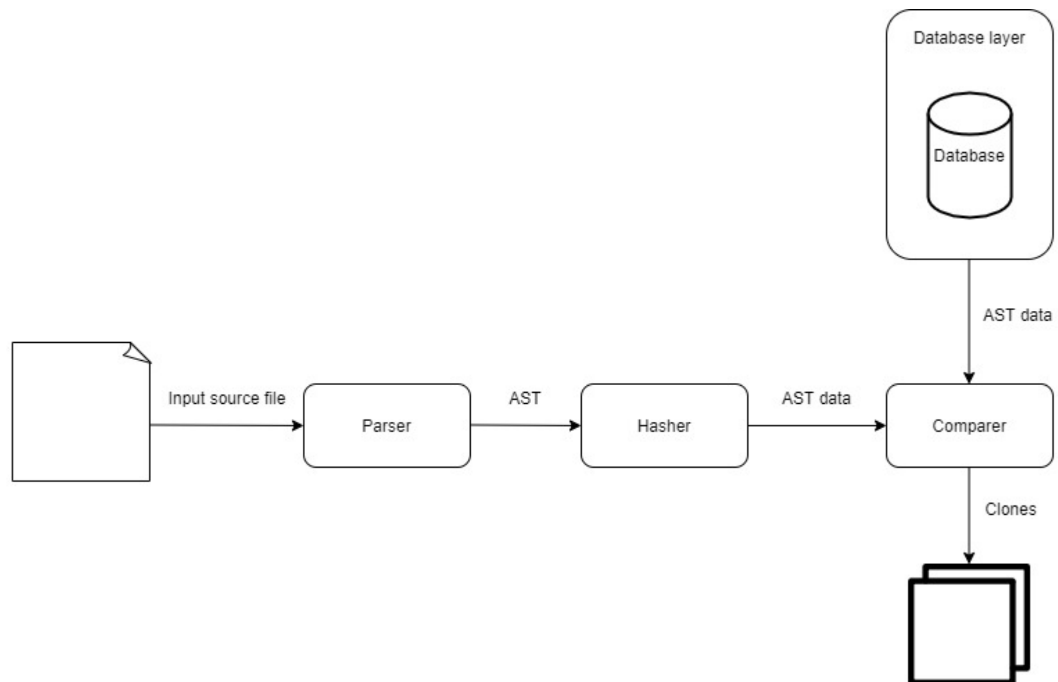

Figure 3.2: Clone detection

Although APIs of individual components are designed to provide the necessary functionality for other components that process their output, it is also possible to use them outside of the pipeline. For example, if there was ever a larger system build on top of the original pipeline, and in some part of this system there would be need to create an AST representing the source file, the *parser* component can be reused for this purpose. Same goes for the *hasher* component and further processing of ASTs[2]. *Database layer* and the *comparer* functionality are too specific to be easily reused, however if suitable scenario was to arise, these components can be reused individually as well.

The system should be language independent. Pipeline should work the same way when finding clones in any programming language[3]. To achieve this independence there needs to be a layer of abstraction somewhere in the pipeline, to shield the langauge independent components from the rest of the system. We decided to place this abstraction in between the *hasher* and the *comparer*. Reason for this comes from the natural properties of the components. While the *parser* is clearly language dependent because it needs to build an AST with structure given by the language grammar, the dependece on the language by the *hasher* is less obvious. Hasher is processing the tree to prepare it for storing in the database or for the comparison. This could be done without the knowledge of the language

[2]However, this is more specific functionality than the parser functionality, which is just to build the AST without any further specialisation.

[3]Clones are identified only in source files written in the same programming language. Detection of cross-language clones is outside of the scope of this work.

specifics, but some information (e.g. roles of the children nodes) would be lost. Making this component language dependent allows the system to extract more data from the source code. On the other hand, the *comparer* does not need to know anything about the language of the source files. It should work with general ASTs to find the clones. The same goes for the *database layer* which does not need to know details about the AST structure.

In following sections we take a closer look at the respective parts of the system in greater detail.

3.1 Parser

The goal of the parser is to create an AST representation of the input source file, which is then used in the rest of the process. AST created by the parser captures all the important information about the source code it represents. This information differs from language to language, but in general every node should contain information about its place in the tree (i.e. reference to the parent node) and information about its position in the source code (i.e. line and column numbers). Other information, like roles of the child nodes, is language specific because it is closely coupled with the language grammar.

When it comes to creating the parser for the specific language, multiple approaches are possible. For simple languages it is feasible to write the parser from scratch. This means processing the source file line by line and building the AST based on language grammar rules. However, this approach is not the best idea when it comes to processing the source files written in the most common general purpose programming languages. These languages have grammars too complex and contain too much syntactic sugar, to be easily processed in this manner.

Some programming languages provide an API to access the AST of the program[4]. Example of such a langauge is Python. When the language runtime is aware of the AST structure, it is possible to serialize the tree. After the tree structure is serialized, the parser can be build in such a way, that it does not process the original source code file but builds the AST based on its serialization. This shields the author of the parser from the concrete syntax of the language. It is easier to parse the structure representation in JSON or XML format, than parsing the source code and deal with all the formatting possibilities of the language. However, this approach introduces another step (exporting the AST from language runtime) to the parsing process.

The most common aproach, often used in big open source projects, is using a generated parser. Parsers generated by tools like Flex [25] and Bison [26] have advantage over parsers created from scratch, because they are easier to implement and less error prone. Similarly to the previously mentioned approach of serializing the AST structure and subsequently parsing it, generating a parser has a disadvantage of adding complexity to the parser via an introduction of external tools.

In this section we did not cover how to create a concrete parser and connect it to the system. For detailed description of the AST structure that should be produced by the parser in order to be processed by other parts of the system see

[4] Either the whole program or ASTs of respective modules.

section 4.2.1.

3.2 Hasher

Pipeline uses hash based approach to identify code clones. We rely on the ability of hash function to avoid collisions in case the subtrees are of different shape. To ensure this ability, the cryptographic hash function should be used. Naturally, after creating an AST representation of the code we need to compute hash values for selected subtrees so they can be used further in the process. The easiest way to achieve this is to traverse the tree and recursively create the hash from the leaves. Details of hashing subtrees were further discussed in section 2.3.2.

However, hash values of subtrees are not the only information we need to extract from the AST. The hash only captures the shape of the tree and tells us which nodes are present in the tree and in what order. Nodes of an AST capture much more information, for example previously mentioned position in the source code, or specifically the identifier nodes and nodes representing numeric values and string constants capture the concrete values that were present in the source code the AST represents. This additional information is as important as the hash value itself, because it helps us to later identify the exact clones, i.e. clones that were created by copy-pasting the code fragments without any further changes (renaming identifiers, modifying string constants, etc.).

Although the computation of the hash value and act of collecting the concrete values from the specific types of nodes are two logically different actions, it makes sense to group them together into one phase of the clone detection process. Reason for this is the fact that traversing of an AST, or any tree structure for that matter, is often achieved with the help of the *visitor pattern*. If we wanted to design aforementioned two actions using the visitor pattern, we would likely use two visitors, one for each action. While this makes sense from the software engineering point of view, in reality it is worth to consider to merge the actions together. If we look closer at the types of the nodes that hold information about concrete values used in the source code of the program, we will find out that these nodes are almost exclusively leaves. Consider the code fragment from figure 3.3 and its AST representation. All the concrete values are located in the leaves themselves or in the lower parts of the tree branches. By merging the computation of the hash and the collection of the concrete values from the nodes, we are able to achieve both goals with only one iteration over the tree. This saves the time that would be wasted by a visitor going through the nodes that do not hold any concrete value information. Drawback of this optimalization is the violation of the *single responsibility principle*.

The hasher is also the last language dependent component in the pipeline[5]. This means that the output needs to be normalized through all the specific hashers, no matter what programming language AST are they able to process. Reason for this normalisation is that components following the processing of the AST (database layer and the comparer) need to work with general AST data, without any dependence on the target language. Details on how is the AST data

[5]It needs to know the structure of the AST, given by the languege grammar, in order to properly traverse the tree.

organized in the output of the hasher are described in chapter 4.2.2.

```
a = 10
b = 20

while a < b:
    a = a + 1

if a > 15:
    print("Value of a is: " + str(a))
```

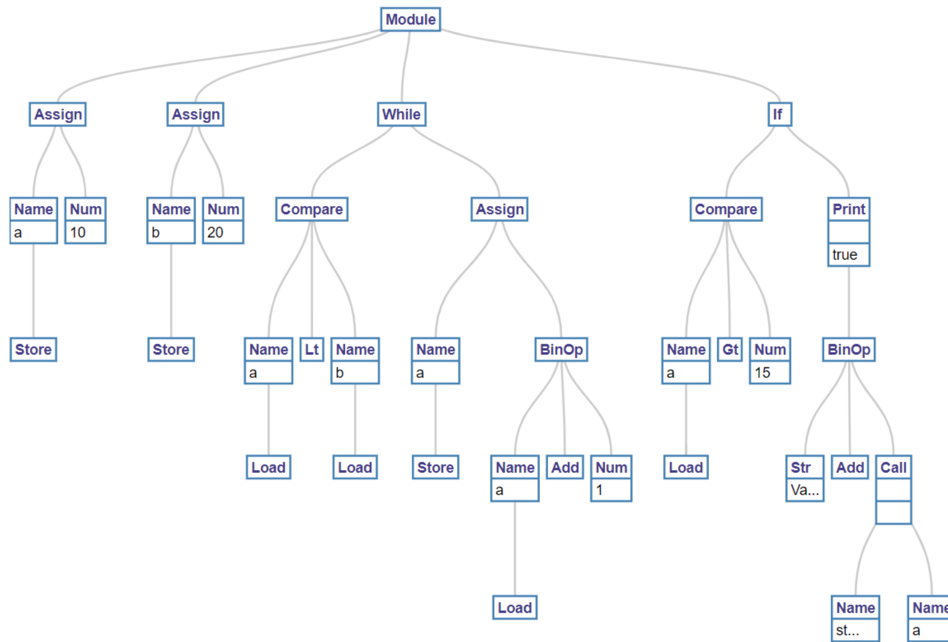

Figure 3.3: Leaves capturing the concrete values

3.3 Database layer

After processing the source files and their AST representations we need to store the collected data for later use. This leads to the use of some kind of permanent storage. By storing preprocessed data, we are able to perform one-to-many comparison faster than if we were to process the files each time there is a new input file submited for clone detection.

The main question when it comes to storing the information about the code is: Which data is necessary for clone detection and therefore needs to be stored in the database? The previously computed hashes of selected parts of ASTs are the core of the clone detection process, which means that this information must be stored. When the comparison is performed we want to localise the cloned code fragment as precisely as possible. This leads to storing of the names of source code files and their association with AST hashes. Also the ability to reconstruct the parent-child relationships between nodes might be needed during the comparison. This information is lost after throwing away the AST and therefore we need to

store this information as well. To be able to localise the clones even better, we can take advantage of the fact that ASTs carry information about the position of code fragments in the file. By storing the positions we are able to precisely show where is the identified clone present. While aforementioned data would be sufficient and the system would correctly find code clones, by adding additional information into the database we might get more details about the clones in the result of the comparison. In the previous section, while discussing the processing of an AST by the hasher component of the pipeline, we mentioned concrete values carried by the leaf nodes. If we store these values, we can enhance the one-to-many comparison process, so it is able to provide the information about identifier renaming as part of its output.

All the necessary data can be stored in a relational database. Structure of such database would not be too complicated. Each subtree could be represented by a record consisting of its hash, name of the file where the original code fragment resides, information about the position, node parent and concrete values[6]. This record might be too complicated to be stored in a single table, so use of standard relational database features, like foreign keys, might be required. It is also possible to use other types of permanent storages. Nature of the data implies that key-value storage could be used as the database. Associating of hash values and file names can be extended to other associations like hash value and position, or hash value and concrete values found in an AST subtree. This leads to key-value pairs, where subtree hash is the key and other information is represented as a value part of the pair, spread across multiple tables based on the value meaning. Structure of the value part depends on the concrete implementation of the used key-value storage. If the database can deal with duplicate keys, the value migh be just a simple record. In case duplicate keys are not supported, the values can consist of multiple records, forming lists of data.

Storage itself should not be directily accessed by other components of the system. To isolate the database we need to create a wrapper that will encapsulate it. This layer must be independent of the input programming language, because we need the database component to be universal. This universality is required by the comparison part of the process. When the comparer requests data from the database it is not performing any further transformations of the data and proceeds straight to the detection of the clones. This is faster than usage of specialised database and further processing of the retrieved data.

Thanks to the normalised output of the hasher component, the only responsibility of the database wrapper is communication with the storage. This covers preparation of database operations (e.g. preparation of SQL statements, creation of database structures, etc.), carrying out the operations themselves and reporting the database errors to the rest of the system as well as recovery from these errors if possible. After retrieving the data from the database, the wrapper must create a view of the data which is then used by the component that requested the data (almost exclusively the comparer component). This view is again language independent. Also other actions that require any database action are part of the wrapper API, i.e. erasing the database or its parts, getting database statistics or dumps of the data.

[6]Of course the unique ID generated by the database would be needed to distinguish between identical code fragments.

3.4 Comparer

The comparer is the module which performs the clone detection. After the input source file is preprocessed by the parser and the hasher, the collected data is not written into the database but instead it is forwarded to the comparer.

Although the system is based on hashing the AST representations of the source code, there are many different options for implementation of the comparer. While the basic functionality of matching subtrees with identical hashes stays the same, the details of the code fragment clones detection may vary. For example, there are multiple options of what the output of the comparer (and the whole system) can look like. One thing that all the options have incommon is that the output somehow describes the clones found in the input file. This information may consist only from simple list of of boolean values for each significant code fragment in the input, which would indicate whether it is a clone of some piece of the code from the database or not. Slightly improved version would include a position of the clone both in the input file and in the file stored in the database[7].

On top of indicating the clone presence, it si possible to provide more information about the clone. Namely, the comparison process might take into an account concrete values of identifiers and literlas. This leads to additional information about the identifier renaming. It can be reflected in the output as mapping of the old names to the new ones, or a similarity metric may be employed to provide the information about how close, in terms of sequence similarity, are the identifier sequences to each other.

AST based methods for source code detection have their limits, as shown by table 1.1. However, if the system attempts[8] to detect the clones created by more advanced scenarios, this logic should be part of the comparer component. In case detection methods became too complex and layered, it is possible to divide the comparer into multiple submodules, where each submodule would perform one part of the clone detection. Of course algorithms implemented as a part of the comparer are again completely language independent and are able to work with general AST data. There is little to no value in introducing input language details this late into the clone detection process.

The fact that the comparer is the last part of the pipeline means that there is a need for an API to be created around the output of the comparer. This API serves as the base for the programs that would use the pipeline as a part of a larger project.

[7]File is already preprocessed and split into AST subtrees.

[8]These attempts would significantly slow down the detection process, see section 2.3.

4. PyPlag

Previous chapter describes the pipeline for clone detection. This pipeline can be implemented to be able to recognize clones in any programming language. As a part of this work we have developed the concrete implementation of the system. We call this implementation **PyPlag**. The comparer and the database layer are implemented as language independent, while the parser and the hasher components are specialised to process Python source files.

Python is currently the only language PyPlag is able to process, but in section 4.3 we provide a guide on how to extend the implementation, so it is able to process other programming languages. We chose Python because of the following reasons:

- Python is language with relatively simple grammar compared to, for example C-like languages like C++, C# or Java. This leads to smaller variety of AST nodes and ultimately reduces the scope of programming work so it can be done as a part of a single thesis. Another benefit of choosing the simpler language is the fact that we can focus more on the clone detection and spend less time on language specific components.

- Python runtime provides a module that is able to examine the program AST. With the help of this module, tools that are able to extract the AST and serialize it can be build. This is an important argument in favor of Python, because implementation of the parser from scratch would be out of the scope of this work.

- In recent years Python became very popular. It is mostly used for artificial inteligence and scripting. According to the GitHub statistics it is the second most popular language on the platform [27]. This means that PyPlag can be applied to the large portion of open source projects right away.

PyPlag implementation supports Python 3.7.3 version.

On top of the pipeline implementation we have also created a frontend in the form of GUI. The frontend is able to set up the clone detection process and present the results. PyPlag pipeline is not closely coupled with our frontend and can still be used separately, the frontend merely builds on the provided API. PyPlag implementation is written in C++. Main reason for this decision was the speed advantage of C++ over other commonly used general purpose programming languages. PyPlag frontend is implemented in Java.

4.1 Python

4.1.1 Overview

Python was created in 1980s by Guido van Rossum. The inspiration for its creation was the ABC langauge, which Python extends by adding new concepts. Implementation of the langauge began in 1989 [28]. Python 2.0 was released in 2000, adding new major features like support for Unicode and improved garbage collector. Another major version, Python 3.0 was released in 2008. This iteration

was a major revision of the language and was not entirely backwards-compatible. This lead to backporting of new features to Python 2.7 version. End-of-life date of Python 2.7 version was set at year 2015, but because of the concern that a big portion of existing code could not be forward-ported to Python 3.0 version, it was postponed to 2020 [29].

Python is a multi-paradigm programming language with full support for object-oriented programming and structured programming. Many of its features support aspect-oriented programming and functional programming. Python's functional programming support builds on the Lisp language and provides *filter*, *map*, *reduce* functions as well as list comprehensions and generator expressions. Other paradigms that are not directly supported by the language are enabled via extensions, for example logic programming [30]. Extensions are crucial part of the language culture, because it was designed to be highly extensible instead of having all the functionality implemented as a part of its core. This makes Python popular as langauge for adding programmable interfaces to existing applications.

Philosophy of the language wa summarized in the document *The Zen of Python* [31], which states 19 core principles that influence the design of the language. These are some of the principles:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Flat is better than nested.
- Readability counts.

Other principles focus on error handling, special cases and implementation guidelines. One of the goals of the developers of Python language is to keep it fun to use. This is reflected in playful approaches to tutorials and reference materials.

Python is interpreted high-level language which leads to worse performance as in the case of non-interpreted languages. This conforms to the langauge philosophy as Python developers strive to avoid premature optimization. When speed is important, Python programmer can take advantage of extension modules written in languages like C and move time-critical code there.

4.1.2 Syntax and semantics

The language has simpler, less-cluttered syntax and grammar and gives developers a choice when it comes to coding methodology. Python is meant to be an easily readable language, which is reflected in use of English keywords instead of punctuation and visually uncluttered formatting. There are no curly brackets to represent beginning and end of the block as in other programming languages and semicolons after statements are optional. Curly brackets are replaced by whitespace indentation. Increase in indentation signalizes the beginning of new block, while decrease in indentation means the end of current block. Therefore, visually the program looks like its own semantic structure. Although most of the other programming languages have formatting guidelines that lead to the same effect, the difference is the fact that this indentation is mandatory in Python.

Statements include standard statements for variable declaration and initialization, control flow and exception handling. Python speciality are generator functions and statements supporting them. These functions allow the programmer to declare a function that behaves like an iterator and can be used in a for loop. Inside these functions the *yield* statement is often used to return a value. With next call to the generator function, the enumeration process is resumed at the position of the *yield* statement. Other statements like *assign*, *function definitions* and control flow constructs like *for* and *while* loops, *break*, *continue*, etc. have same meaning as in other commonly used programming languages. Complete Python 3.7.3 grammar can be found in appendix A.

Python expressions are similar to C-like languages, but there are some exceptions to this rule. Basic aritmethics works as in other languages, but there are two kinds of division. The integer division results in integer value and floating point division results in floating point value. These are separete because of Python type system, which is dynamic. Python also supports wide range of mathematical expressions. For example, there are operators for exponentiation, matrix multiplication and cross product of two vectors. The language supports anonymous functions implemented as lambda expressions. These are more limited than in other languages and allow only one expression in the body of an anonymous function. Another Python speciality is *sequence unpacking*. This feature allows to put multiple assignable entities (i.e. variables, properties, etc.) on the left side of an assignment statement, which then expects iterable object on the right side. Values are assigned to the left side entities in the order specified by iterable object. Number of values created be iteration must be equal to the number of left side expressions. Python supports *array slicing* expressions as a way to index lists. Slices are using following syntax

a[**start:stop**] or **a**[**start:stop:step**],

which results in taking elements from *start* index up to the *stop* index. The *step* parameter allows to periodically skip some elements or reverse the order of enumeration. The last language specific concept worth mentioning is the support of multiple kinds of string literals. In Python these strings are allowed:

- Strings enclosed in single or double quote marks. These strings use the backslash as an escape character, similarly to other languages.
- Triple-quoted strings, i.e. string value is delimeted by ’” sequence at the beginning and the end of the string. They may span multiple lines.
- *Raw strings*, denoted by prefixing the literal with an **r**. This notation disables escape sequences.

In Python, a distinction between expression and statements is enforced. This leads to redundant functionality of some concepts, for example *list comprehensions* and *for* loops. Statement cannot be a part of an expression, therefore the previously mentioned anonymous function restriction. This separation of statements and expressions results in avoiding programming mistakes, because the mistake is deemed as syntactic error. Classic example of such an error would be mistaking an equality operator (==) for an assignment operator (=) in a condition expression.

4.1.3 Typing

Python uses duck typing, i.e. an object's suitability is determined by its properties, not by the type of the object. Type constraints are not checked at compile time, but instead operations may fail on an object of incorrect type. Python is dynamically typed language but the number of implicit type conversions is still smaller than in other languages. Operations that are not well-defined are forbidden. Example of such operation is addition of string and a number.

The language allows user defined types (classes). All user defined classes are instances of the *metaclass*. This concept introduces metaprogramming and reflection to the language. Classes are allowed to contain functions, called methods. Methods are attached to class and may be called on instances of the class.

The long term goal of the authors of the language is to support gradual typing. Gradual typing is a type system in which some expressions are given types and the correctness is checked at compile-time, i.e. static typing, but other expressions are left untyped and type errors are reported at run-time, i.e. dynamic typing. This concepts allows to choose either paradigm from the single language, depending on the context in which it is used. Current state of Python type system is shown in figure 4.1

4.1.4 Uses

Python is consistently ranked as one of the most popular programming languages in the TIOBE Programming Community Index [32]. An empirical study found that scripting languages, like Python, are more productive for problems involving string manipulation and dictionary search, than conventional languages [33].

Python can serve as a scripting language for web applications [34]. Tools like Pyjs [35] can be used for development of client-side of Ajax applications. Python is also part of tools used as data mappers for relational databases [36]. Libraries NumPy, SciPy and Matplotlib allow effective use of Python in scientific computing. It has also been commonly used in artificial intelligence projects and natural language processing. Information security industry also extensively uses Python.

Many operating systems include Python as a standard component. For example, the language is part of Debain, OpenSuse, Fedora, FreeBSD and macOS. Python can be used directly from the command line. Linux distributions supporting the language also use installers written in Python.

Because of user-friendly concepts of Python as well as the fact that it is easy-to-understand language, it is commonly used as an intro language in computer science and programming courses. It allows students to easily learn programming concepts, which they can then apply to other programming languages.

Philosophy and design of Python have influenced other programming languages. ECMAScript uses generators from Python and Cobra uses indentation and similar syntax. Motivation behind Groovy language was to bring Python design philosophy to Java, and Apple's programming language Swift adopted some of Python's syntax.

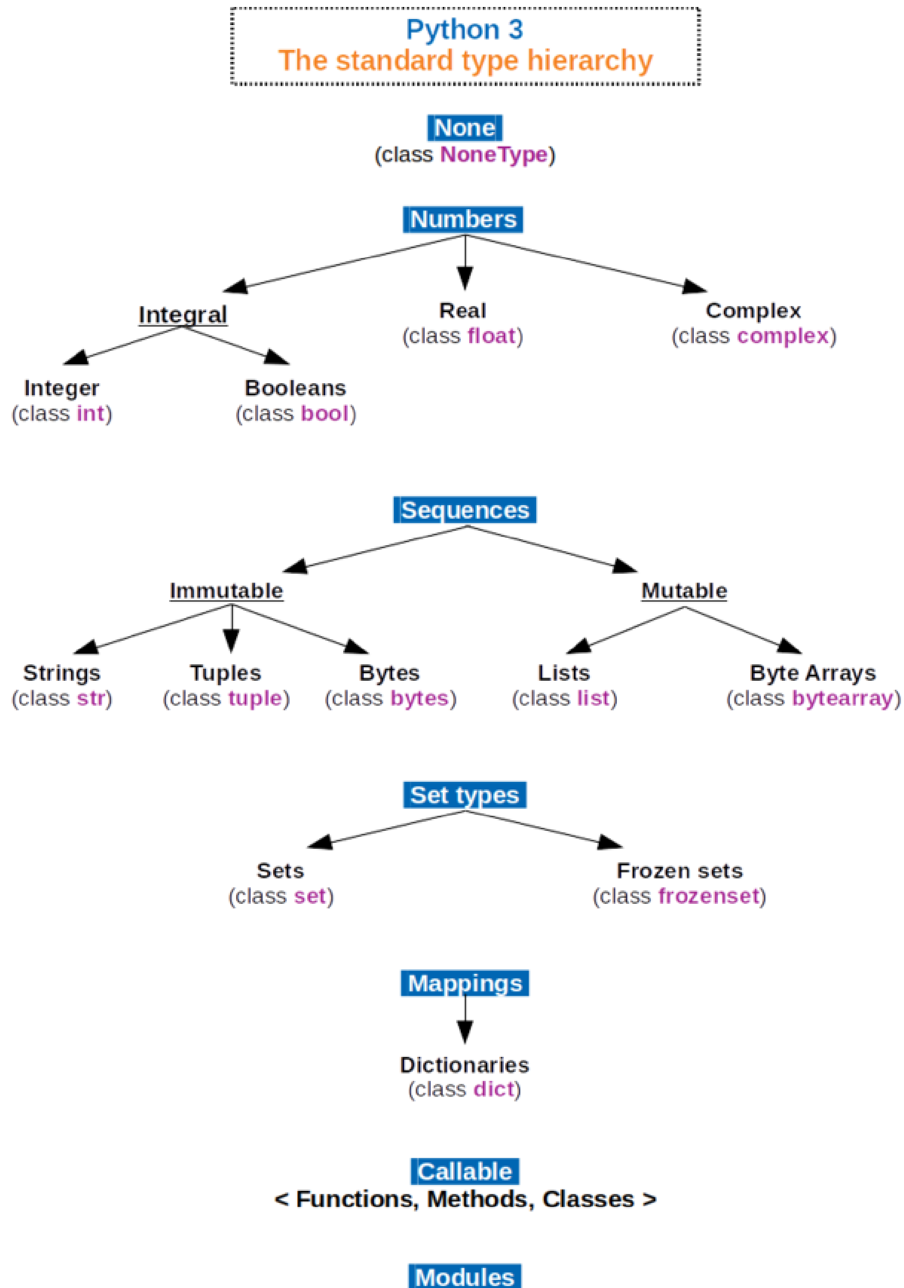

Figure 4.1: Python 3 types [37]

4.2 Implementation details

In this section we describe technologies, algorithms and tools which are incorporated in PyPlag. Respective subsections follow the pipeline components division from chapter 3. On top of the pipeline implementation description, the frontend GUI is also covered in this section.

4.2.1 Parser

Rather than implementing the parser from scratch, we instead take advantage of Python **ast** module [38]. This module provides an API for AST inspection and manipulation. Python is able to provide information about the tree nodes, such as the type of the node and properties of the node, like the node descendants and the node parent. On top of this module the *astexport* tool was developed as an open source project by Poli [39]. This tool is used to serialize Python ASTs into JSON. JSON representation of an AST consists of objects representing the nodes in the tree. These objects can contain either other objects and lists of objects representing child nodes of the node, or other properties captured by the AST node it represents. Each node object contains at least the node name property, the string property holding the name of the node as specified by Python grammar. Statement and expression nodes also always carry information about their position in the source file via line number and column offset integer properties. Examples of JSON serialization of Python ASTs by *astexport* tool can be found in appendix B.

The nature of JSON format used by *astexport* leads to the construction of the tree in preorder traversal manner. By reading the JSON file as a stream of characters, we always encounter the parent node first, followed by the children nodes and so on. Another approach to building the tree would be *postorder traversal*. This would require building the tree in the bottom-up manner. First the forrest of leaf nodes would be created, and then in each step a group of nodes would be connected to their parent. After the last step, there would be only one tree in the forest and that would be the AST we wanted to construct. However, the bottom-up approach is not well suited to be used in our case, because leaf nodes are scattered all around the JSON serialization of AST. There is no easy way of identifying the leaf node while parsing the data. Therefore we decided to use the straightforward approach and build the tree from the root.

The parser implementation itself consists of methods that are responsible for parsing the JSON node objects and their properties. Based on the parsed data the AST is build. Objects created at runtime, representing respective AST nodes, copy the hierarchy described by Python grammar. Each type of AST node is represented by different type of the node object. All node objects are derived from single base class representing general Python AST node. There are multiple degrees of derivation between the general node and concrete nodes, further dividing the nodes into multiple categories, like statement and expression nodes. This approach is more suitable for our use of AST than the approach using general nodes. If the AST was represented as a structure constructed solely from one type of a node, we would run into problems later in the pipeline. For example while hashing the tree, we need to incorporate the role of a node into the hash value. With one general type of a node, the children of the node would be all grouped into one collection and distinguishing between node roles would become complicated. Also not every node has the same properties. For example nodes representing binary operations carry information about the concrete operation, while literal nodes need to be able to hold a value of a literal. Grouping all possible properties of a Python node into one object would be unnecessarily difficult to follow while working with the code.

When there is enough information extracted from the stream to determine

which node object should be constructed, the parser calls one of its *factory methods* which creates the desired object. The node is subsequently connected to the rest of the tree and parser continues to examine the stream data. The result of parsing process is AST of the input Python program, with all the information specified by Python grammar.

Above described parsing process is implemented as a part of Pyplag. This implementation is not used directly, but via interface it implements. If there was ever need to use different parser implementation, it can be done so without touching the rest of the system. The interface is specific to Python language and its grammar, but it is not tied with JSON format of serialization. Other formats for AST serialization may be used as well, but for PyPlag's purposes, use of JSON is feasible.

Parser is the first component of the pipeline and therefore is responsible for interaction with input of the system. The side effect of using the *astexport* tool is that the input of the system is not the Python source file itself. All source files presented to the system must be first preprocessed by *astexport* which produces JSON files that can be then used as a system input. It takes *astexport* approximately 200 ms to preprocess the file resulting in an AST with one thousand nodes. Additional 53 ms are needed to parse the serialized tree. Hashing of the same tree takes approximately 1.52 ms, implying that the input parsing is the bottleneck of the entire pipeline.

4.2.2 Hasher

After an AST is build, PyPlag needs to prepare it for comparison. The tree is traversed and during the traversal hashes of subtrees are calculated and the data necessary for comparison is extracted from the nodes. Not every subtree is converted to its hash value. Because not every subtree needs to be compared or stored in the database, there is a size limit passed to the hasher as input parameter. Only subtrees that are bigger, with respect to the number of nodes, than the size limit are hashed. This is done because not all code fragments and their respective subtrees in AST are significant enough to be considered during the comparison.

Subtree hashing

The hash for all subtrees with size greater than the hasher size limit is computed within the single traversal of the tree. We are using preorder traversal, during which the string representation of the tree is created. Each time the new node is encountred, string is extended by the node name and its role in an AST. If the subtree rooted at the node satisfies the size limit, the part of the string representing the subtree is hashed and stored for later use. The process can be summarized by algorithm 1.

Hashing of the string representation of the subtrees is done with the help of **MD5** hash function. This function produces 128-bit hash values [24]. With 2^{128} different possible hash values, we can hash 2^{64} subtrees before the chance of a collision reaches 50%. Together with preorder traversal and incorporation of the node roles in the AST, the MD5 hash value captures the shape and the composition of the tree.

Algorithm 1: Subtree hashing

HashSubtree (N, S)
    inputs : Subtree root N; tree string S
    outputs: tree string S; hashed subtrees *hashes*
    $begin \leftarrow Length(S)$;
    $S \leftarrow S + role\ of\ N$;
    $S \leftarrow S + name\ of\ N$;
    foreach *node* $C_i \in$ *child nodes of* N **do**
        $HashSubtree(C_i, S)$;
    $end \leftarrow Length(S)$;
    if *size of subtree* $N \geq$ *size limit* **then**
        $hashes \leftarrow hashes \cup Hash(Substr(S, begin, end))$;

Use of the size limit as an indication whether the subtree hash should be stored or not has its limitations. Consider the subtree that satisfies the size limit as a whole, but every subtree rooted in the child node of the root does not satisfy the size limit. Now consider another subtree which was created by cloning the original subtree and additional child nodes were inserted before the first original child node. Hash values of these two subtrees will be different. This is correct behaviour but we are losing a big part of the information about child nodes, because they do not satisfy the size limit. To solve this we need to store hashes of sequences of nodes which are not stored individually.

Sequence hashing

Sequence hashing needs to be executed at every place of the program where a *body* is allowed. Code clones may be a part of the body, but the parent node of the body is node the clone itself. Statements in the body are too small to be considered significant and it does not make sense to detect them as clones. For example, if we checked every statement below the size limit for match in the database, then every variable declaration could be considered a clone. This is wrong, as variable declaration by itself does not carry enough semantic context to be proclaimed a clone. However, if we group multiple statements together, they might be significant enough to be a clone. The number of statements in the sequence will vary from language to language but in principle, the sequence should be long enough to be semanticaly significant but not too long, because then the condition would be satisfied only for large sequences and majority of clones would not be detected. After the sequence is detected, it is hashed and stored in the same way as single node hashes. Hashes of sequences can be mixed with single node hashes because our hashing function is strong enough, so the collisions are unlikely. Algorithm 2 describes the process in greater detail.

Mapping AST data to subtree hashes

Hasher is the last component involved before the comparison of the trees or before storing the trees in the database. This means that it needs to prepare all

Algorithm 2: Sequence hashing

```
HashSequence (B)
    inputs  : Body of statements B
    outputs: Hashed sequences hashes
    S ← empty string;
    count ← 0;
    foreach statement S_i ∈ B do
        if size of S_i < size limit then
            HashSubtree(S_i, S);
            S ← S + sequence marker;
            count ← count + 1;
        else
            if count ≥ sequence limit then
                hashes ← hashes ∪ Hash(S);
            S ← empty string;
            count ← 0;
```

the needed data about AST nodes for later use. Besides computing the subtree and sequence hashes, this involves:

- Mapping the information about parent-child relationship of nodes to respective hash values. This information can be later used to reconstruct the tree, or its parts, after reading hash records from the database.

- Mapping the values of literals to respective hash values. Comparison of concrete values of identifiers or numeric and string constants is a part of the comparison of two subtrees.

- Mapping of line numbers and column offsets to respective hash values. Position of the code fragments in the source file is useful for pinpointing the location of a clone as a part of the output.

Mapping of tree edges and positions of hash values is done in straight forward manner. During the traversal of the tree, after the hash of a node is computed, it is stored in the key-value container, where key is the pointer to the node and the value is the hash. Then, after the computation of hash values is finished, desired information is easily and quickly accessed. Information about the position of the code fragment is stored as a property of the AST node and we can access it via the node pointer. Similarly the parent hash is found. First the parent is accessed via the node pointer and its hash value is looked up in the table once we obtained the pointer. The creation of mapping of literals to hash values is more complicated. Concrete values are typically stored in the leaf nodes and by definition these nodes are too small to satisfy the size limit and be stored in the database. Therefore we cannot associate literal values with the nodes they are present in. Instead we associate them with the whole subtree.

Disadvantage of this solution is the fact that we lose the information about the exact location of the literal. Loss of this information can be solved by taking into an account the order of literals in the list associated with the subtree. If

two subtrees have the same hash (and therefore the same shape), they will have the same number of literals as well. Then the order of literals determines if the subtrees are identical or not.

Naturally the subtree that literals are associated with is the smallest one that satisfies the size limit. This means that after encountering the node that represents a literal during the traversal, the nodes on the path from the node to the root are analyzed and the first one that satisfies the limit is selected:

Algorithm 3: Literal-Subtree association

LiteralSubtreeAssociation (N)
  inputs : Literal node N
  outputs: Updated literal mapping *literals*
  $C \leftarrow N$;
  while *(C **is not** root) **and** (size of C < size limit)* **do**
    $C \leftarrow parent\ of\ C$;
  $literals[C] \leftarrow literals[C] \cup literal\ stored\ in\ N$;

Storing literal values in this manner leads to different mappings for different size limits. The hasher size limit is passed to it as an input parameter from the user. If the user has used different size limit for hashing subtrees while creating the database and different size limit while comparing a source file with the database, PyPlag will not be able to correctly compare literal values. Example of such a situation can be seen in figure 4.2. Literals are mapped to different subtrees and therefore there is no straightforward way to compare the lists. Figure 4.2 only shows the mapping of identifier names. In reality, separate mapping for numeric and string values would be created.

We described each algorithm used in the hasher separately so they are easier to understand. The implementation interleaves the steps of all algorithms so all the information can be collected in one traversal of an AST. After the computation is done, information about subtree hashes and mapping of AST data can be accessed through API of the hasher by other components of the system.

4.2.3 Database layer

Database layer does not hold any logic used for AST processing or comparing. Its only responsibility is to store the information passed to it. Permanent storage we chose for this purpose is LevelDB by Google [40]. LevelDB is a fast key-value storage library, mapping string keys to string values. In this context a string is considered an arbitrary sequence of bytes.

The decision to use a key-value storage instead of a traditional relational database was made while designing the mapping of AST data to tree hashes. Every mapping is represented in memory as key-value container. This representation is simple and easy to work with. By using the same format for permanent storage, we minimize the amount of further processing of the data in preparation for writing to database.

After an AST is processed by the hasher, the data is accessed via hasher API from the database layer. Before the data is written into the permanent storage,

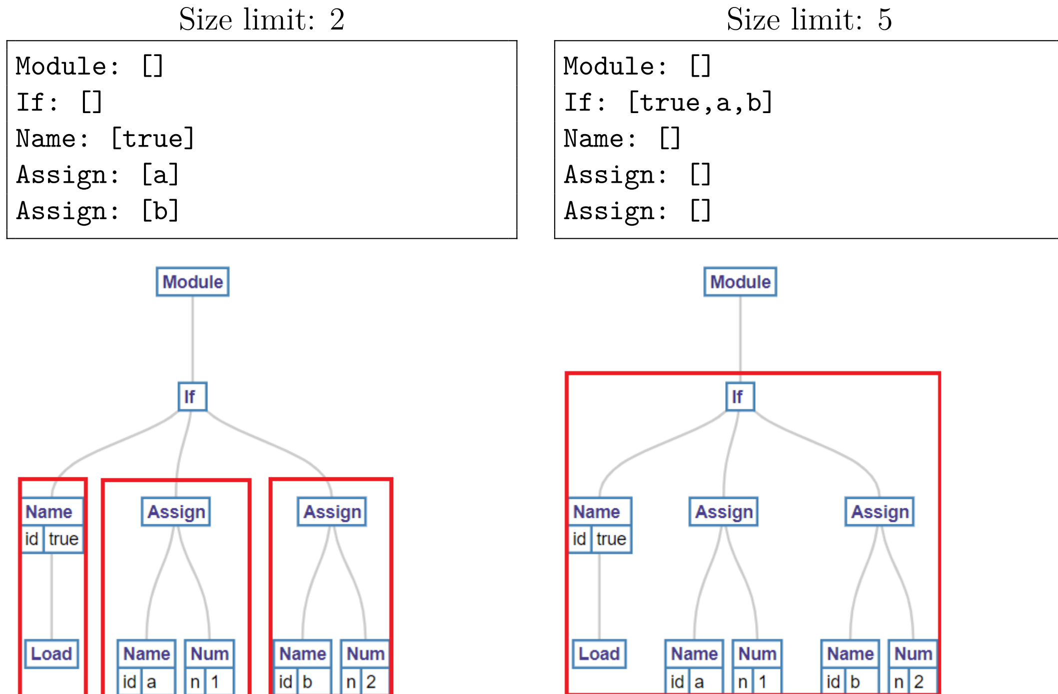

Figure 4.2: Effect of different size limits on literal mapping

the LevelDB database is opened. The library creates a folder at specified location. This folder then holds all metada created by the database as well as the data itself. Leveldb does not support multiple key-value tables in a way in which all the tables are aware of other tables in the database. This means that if we need to use multiple tables, each table needs its folder. Each mapping created by the hasher has its own table in the permanent storage. Folders are created in the directory where the Pyplag executable resides in. Names of the folders correspond to the data they store:

- **tree** - mapping of subtree hashes and parent hashes,
- **position** - mapping of positions and subtree hashes,
- **identifiers** - mapping of identifier literals and subtree hashes,
- **numbers** - mapping of number literals and subtree hashes,
- **strings** - mapping of string literals and subtree hashes.

The difference between records created in memory by the hasher and permanent storage records is in the fact that input files are processed sequentially. This means that while the hasher is traversing the AST, we know at every moment to which file the AST belongs. But in the database all the data is mixed together. Because the key-value tables are indexed by subtree hashes, the records with the same key must be concatenated into the single value. To distinguish between records from different files, the file name is part of the database record. File name is also important later, when the clones are identified.

After the LevelDB tables are opened, the data is prepared to be written in the database. Records are processed sequentially and the process is the same

for all mappings created by the hasher. First the list of values is converted into string, because LevelDB does not support structured values. Then the table is asked through LevelDB API if there is already a record with the same key. If not, the key-value pair is inserted, otherwise the value is loaded into the memory and a new record is appended to the value, before writing the value back into the database. Structure of permanent storage records is described in table 4.1. Symbols + and * have their usual regular expression meanings, symbols [and] enclose the key value pair and symbols (and) enclose respective parts of the value. In the tree table there might be a record with an empty value. This means that this hash value represents entire tree and therefore there is no parent the hash should be associated with.

Algorithm 4: Write data record to the database

WriteRecord (K, V, F)
- **inputs** : Record key K; record value list V; file name F
- $S \leftarrow$ *empty string*;
- **foreach** *value* U *in* V **do**
  - $S \leftarrow S + U + \textit{delimeter}$;
- $S \leftarrow S + F + \textit{delimeter}$;
- **if** *table contains key* K **then**
  - $\textit{value} \leftarrow \textit{Load}(K)$;
  - $\textit{value} \leftarrow \textit{value} + S$;
  - $\textit{Store}(K, \textit{value})$;
- **else**
  - $\textit{Store}(K, S)$;

Table	Record format
tree	[hash, (hash+, file)*]
position	[hash, ((line, column)+, file)+]
identifiers, numbers, strings	[hash, (literal+, file)+]

Table 4.1: LevelDB tables structure

Reading values from the database is straightforward. Values are parsed and the data is encapsulated in a structure depending on the type of the mapping it represents. This structure is then returned from the database layer to the caller.

LevelDB performance

LevelDB documentation provides information about the performance of the library. For our purposes the most important benchmark for writing the data is the *overwrite*. This benchmark indicates the speed of updating the existing keys in the database [40]. This operation is used most often when writing into our database. When it comes to reading we are most interested in the *random lookup* benchmark, as there is no way to predict which subtree is going to be looked up

next. Table 4.2 provides information about the machine on which the measurements were executed and information about the experiment parameters. Table 4.3 presents the values of aforementioned benchmarks.

CPU	4 x Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz
CPU Cache	4096 KB
Keys	16 bytes each
Values	100 bytes each
Entries	1000000

Table 4.2: LevelDB benchmark setup [40]

Operation	**Speed**
Overwrite	2.380 micros/op
Random lookup	16.677 micros/op

Table 4.3: LevelDB benchmark [40]

4.2.4 Comparer

Detection of clones is the last action in the pipeline and its result forms the result of the entire system. The process can be divided into three phases:

- Retrieving clones of single subtrees from the database.
- Retrieving clones of sequences from the database.
- Identifier analysis.

After these phases are completed, all source code clones present in the input file should be identified. Output of the comparer is a collection of objects describing the group of clones for each node in the input AST. Single output object consists of:

- Pointer to the original node in the AST.
- A number representing the result of identifier analysis, for each clone. Number is associated with the name of the file in which the clone is present.
- List of positions of clones in their respective files.
- List of identifier literals present in clones.
- List of identifier literals mapped to the original AST node.
- In case the original node is starting point of a sequence clone, the sequence is represented as list of pointers to sequence nodes.

Structure of the output was designed to be used to visualize the output of PyPlag. However, futher analysis of detected clones can be build on top of the pipeline output as well.

Single subtree clones

After recieving AST data of the input source file form hasher, the comparer needs to identify all clones of code fragments from the input file the system had previously seen. The first and the most important step to identify a clone from the database, is comparing the hash of in-memory node to the hashes in the database. Because of the properties of our hash function, the chance of false positives is very low and the equality of hash values implies the same shape of AST subtrees. Identical shape of subtrees is good indication of a clone. However, we still need to take a look at identifiers present in code fragments to find out to what extent the code was copied. This will be done in the identifier analysis.

The comparer is not traversing the AST again after it was processed by the hasher. Instead it recieves the mapping of the tree nodes and their hash values. By iterating over this collection, a single database query is able to retrieve all the records with the same hash. AST nodes with the hash which is not present in the database are filtered out and no longer taken into the consideration during the clone detection. For the rest of the nodes, additional queries are performed to retrieve the information about the positions and literals associated with the clones. After all the data is gathered from the database, the remaining nodes are sent to the identifier analysis to determine the level of similarity between them and their clones based on the identifier values.

Algorithm 5: Get subtree clones from the database

```
GetSubtreeClones (S, H, D)
    inputs : List of input AST subtrees S; hashes of input AST subtrees
             H; PyPlag database D
    outputs: List of subtrees with hash present in the database result;
             additional subtree data from the database data
    result ← empty set;
    foreach node N in S do
        if H[N] is in D then
            result ← result ∪ N;
    foreach node N in result do
        data[N] ← LoadDataFromDB(H[N]);
```

Sequence clones

Sequence clones are handled similarly to single subtree clones. First, the database is scanned for sequence hashes present in the input AST. The difference between the subtree clones and sequence clones is in detection of the largest sequence clone. The hasher prepares multiple overlaping sequences, so we are able to detect cloned parts of the sequence of statements as well as the entire sequence. After the sequences without the clones in the database are filtered out, the comparer checks first node of every remaining sequence and selects the longest sequence starting at this node. Then the additional information about the sequences is retrieved from the dabase the same way as in handling the subtree clones.

Identifier analysis is not applied to sequences. When the hasher creates the mapping of literals to tree nodes, only the nodes over the size limit are used. As we mentioned in the hasher section, by definition the sequence nodes are not big enough to be used in mapping. Therefore there is no easy way to reconstruct the identifer placement when loading the data from the database. This could be bypassed to some extent by looking at the parent node of the sequence and using the identifiers mapped to this node. This approach would only work for entire sequence clones and not for subsequence clones, as the identifier lists would be misaligned, due to the different shape of the subtrees. If the entire sequences of statements share the shape of ASTs, then there is a very high chance that they have the same semantical meaning. Heuristics based on the shape of trees is in this case enough to proclaim sequences as clones.

Sequence clones are handled after the comparer has finished processing the subtree clones. Output object for each sequence clone is added to the output of the comparer.

Algorithm 6: Get sequence clones from the database

```
GetSequenceClones (L, N, H, D)
    inputs  : List of sequences S; List of starting nodes of sequences N;
              hashes of sequences H; PyPlag database D
    outputs: List of sequences with hash present in the database result;
              additional sequence data from the database data
    subResult ← empty set;
    result ← empty set;
    foreach sequence S in L do
        if H[S] is in D then
            subResult ← subResult ∪ S;
    foreach node M in N do
        result ← result ∪ GetLongestSequence(M, subResult);
    foreach sequence S in result do
        data[S] ← LoadDataFromDB(H[S]);
```

Identifier analysis

After getting the subtrees with identical shape as subtrees from the input AST, we need to analyse other semantic properties of the clones. The identical shape does not automatically mean that the subtrees are representing cloned code fragments. This holds true especially for smaller pieces of code, like variable declarations and simple expressions. Example of such situation was mentioned in section 2.2.2, where the two pieces of code had the same AST structure with different identifiers. In this case the identifier values change the meaning of the code. To be able to detect this change, we need to compare the identifier sequences mapped to these subtrees. By replacing the identifier names with numbers, we bypass identifier renaming modifications and all that is left is to check the order of identifiers in the sequence.

Algorithm 7: Replacing identifier sequence with sequence of numbers

IdentifierNumbering (S)
  inputs : Identifier sequence S
  outputs: Sequence of numbers representing identifier order $numbers$
  $numbers \leftarrow empty\ list$;
  $counts \leftarrow empty\ map$;
  $number \leftarrow 0$;
  foreach *identifier I in S* **do**
    if *I is not a key in counts* **then**
      $counts[I] \leftarrow number$;
      $number \leftarrow number + 1$;
    $numbers \leftarrow numbers \cup counts[I]$;

To check if the sequences are identical, we compare the numbers one by one. In case the sequences are not identical, or they have different lengths as a consequence of different subtree limit paremeter used while creating the identifier mapping in the hasher, we need to represent this fact in the result. While identical identifier sequences indicate that the code fragments are clones of each other with possible identifier renaming, sequences with some differences only indicate the same structure of the code. We keep subtrees with non-identical identifier sequences in the result to highlight identical constructs in the code. This information can be interpreted by the user as a place in the code which might be replaced with a parametrized subroutine. To distinguish between clones and fragments with identical structure we use the number representing the similarity of identifier sequences. This number is computed for two sequences with lengths M and N as:

$$\frac{2 \cdot t}{2 \cdot t + (M - t) + (N - t)},$$

where t represents the number of identifiers with the same ordering number at the same position. Result of this formula is 1 if the sequences are the same length with identical identifiers at the same positions. The result is a number between 0 and 1 if the sequences are non-identical.

The same approach can be used for comparing other literals. However, string and number constants are not as interesting as identifiers from the semantics point of view. The constant does not realy change the meaning of the code eventhough it might change the result of the computation. PyPlag is prepared to extend literal sequence comparison to strings and numbers, as these values are stored in the database as well.

After the comparer is done with all the processing of the input AST and database data, the result capturing the information about the source code clones is returned to the caller through the PyPlag API. Documentation of this API si part of section 4.4.

4.2.5 Frontend

To create an user interface for PyPlag we decided to use GUI (Graphical User Interface) over the command line interface. Main reason for this decision was the number of inputs for the PyPlag pipeline. There are two different use cases of the system, first being the building of the database and the second is detection of clones in the input source file. In case of building the database, the system usually needs multiple source files and providing information about the location of the files via the command line might be tedious. Another problem with the command line interface would be result presentation. There is no straightforward way to represent the information gathered by PyPlag on the command line.

GUI is implemented in Java, while the PyPlag pipeline is written in C++. Decision to involve another language in the project came down to the support of Java for creating applications with graphical user interfaces. The complexity and the amount of work required would be far greater if we were to use any of C++ libraries instead. The input and output data is transfered between GUI and the pipeline through JNI (Java Native Interface).

Responsibility of the frontend is to make the work with PyPlag easier for the user. It does not process the data in any way, but it automatizes the export of Python ASTs to JSON format, so they can be later parsed by the pipeline. Handling the input for both use cases of PyPlag and presenting the results of clone detection is also the responsibility of GUI.

Forwarding the input data through JNI is done via calling methods from library to which C++ code was compiled. Handling of the output is more complicated. The collection of objects created by the comparer as the result of clone detection is too complex to be easily mapped to Java objects and handled at the fronted. Instead we have chosen to extract the information from the result at the C++ side and send it back to the frontend as an array of strings. The array is then processed by the fronted and the results are shown to the user. Figure 4.3 shows the communication between the frontend and the PyPlag pipeline.

Instructions for work with GUI are covered in section 4.4.

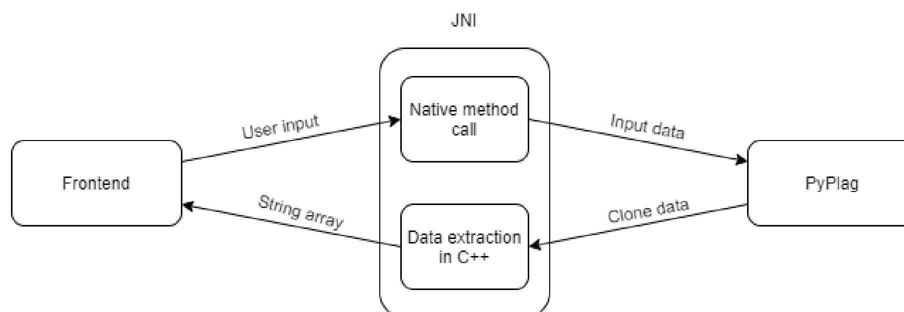

Figure 4.3: Communication between frontend and PyPlag

4.3 Extending the system

To add support for other languages to PyPlag, only the language dependent parts need to be implemented (parser and hasher). The rest of the pipeline does

not work with language specific parts and as long as the interfaces of language dependent parts are functioning as intended, the process of clone detection will work as described in previous sections.

Parser

There is no common interface for the parser component. The only requirement on the parser is to produce an AST constructed from objects that inherit from base AST node class. This base class carries only the basic data, shared across ASTs of all commonly used programming languages:

- Pointer to the parent of the node in an AST.
- Mass of the subtree rooted in the node. Mass is the number of nodes in the subtree.

There are also methods for accessing the line number and column offset of the language construct represented by the node. By default they return -1, but it is expected to override them to function properly.

Implementation of the parser for languages with more complex grammar than Python might require use of additional tools to preprocess the source files, or to generate the parser itself. We discussed alternative approaches for parser implementation in section 3.1

Hasher

The hasher implementations for additional languages must inherit from the base hasher class. This class defines an interface used by the rest of the pipeline to access the data the hasher has extracted from an AST. Base class does not define any data structures for internal works of the hasher.

Methods from the hasher interface can be divided into two groups based on which component of the pipeline uses them. There are two components of the pipeline which use the hasher. When the database of subtrees is being build, the database layer writes the data accumulated by the hasher into the database. These methods all return the mapping of the data to the subtree hash (described in 4.2.3). Container used to represent the mapping is **std::unorder_map** with string key (the hash) and value depending on the type of the data:

- **std::vector<std::string>** for both the parent hash mapping and literal mapping,
- **std::vector<std::pair<int, int>>** for the mapping of the positions. The first number in the pair is the line number and the second one is the column offset.

Data used by the comparer are accessed via methods returning **std::unordered_map** with a pointer to the AST node as the key and the values:

- **std::string** for mapping of the nodes to their hashes and for the mapping of the hashes of sequences to the first node of the sequence,
- **std::vector<std::string>** for mapping of the literal values to subtrees,

- **std::vector** of node pointers. This represents the mapping of the statement node sequences to the first node of the sequence.

On top of the methods for accessing the data, there is one more method in the interface. This method is used to invoke the computation carried out by the hasher. The method does not return any data, but after it returns to the caller, all the processing of an AST should be finished. At this point the rest of the methods from the hasher can be called and they should return correct data.

After the implementation of language specific components is done, we need to connect them to the rest of the system. The PyPlag pipeline has API that allows the user to perform the most common actions, like creating the database, hashing trees and finding clones. Class on which these methods can be called needs to know the language that will be processed by the pipeline upon the initialization of the object. Based on the information about the language, the appropriate hasher is created. This hasher is then used for processing ASTs throughout the lifespan of the API object. In places where the parser is used, the new parser needs to be included as an option for the respective language.

More information about the pipeline API and on how to use it, can be found in the following section.

4.4 User guide

4.4.1 Build

Prerequisities

First we need to install *astexport* tool, so we are able to preprocess the python source files and create the input for the pipeline. The tool can be build from source, which is available at [39] and also is a part of the electronic version. For simpler installation it is possible to use Python package installer *pip3*. Using *pip3* the tool can be installed by using the following command:

pip3 install astexport

Make sure that *astexport* is part of *PATH* environment variable. The tool uses Python *ast* module to extract an AST of a program. Structure of ASTs vary between different versions of the language. PyPlag parser is able to process serialization of Python 3.7.3 source files. Both older and newer versions (namely 3.6 and 3.8) of Python use different AST structure. To use PyPlag properly make sure that the correct version of Python is installed on the machine, i.e. version 3.7.3 which was the newest release at the time the parser was being implemented.

Another tool that needs to be installed is *git*. PyPlag frontend uses *git* to clone repositories from input URL. We do not include a guide for *git* installation as it is a widely used tool.

To compile the C++ part of the source code user needs *gcc* 7 compiler and *make* tool. Fronted requires Java JDK 11, *JavaFX* library and *ant* for compilation. Environment variable *JAVA_HOME* needs to be set correctly before attempting to build PyPlag.

Used software

PyPlag uses multiple tools in addition to its own source code. Pipeline itself depends on *astexport* tool and *LevelDB* library. These parts of the system need to be properly installed/build for the system to function properly. While the installation of *astexport* is straightforward, use of *LevelDB* is more complicated. The library does not provide any option for system installation and can be used only as a C++ library. Therefore it needs to be build from source and linked to PyPlag. Use of JNI further complicates the use of *LevelDB* because it needs to be built as a shared library and its path must be added to *LD_LIBRARY_PATH* environment variable. Source code of both tools is a part of an electronic version of this work.

Another part of the system that was not developed originally for the PyPlag is the source code of MD5 hash function. Used hash function implementation is available at [41] and the code is also a part of the electronic version of this work.

Building PyPlag

PyPlag source code is available at [42] and it is also part of the electronic version of this work. To build PyPlag follow these steps:

1. **tar -xvf thesis_attachments.zip**
2. **cd thesis_attachments/src**
3. **. ./configure**
4. **make**
5. **cd pyplagGUI**
6. **ant -Djavafx.path="*path-to-javafx-lib-folder*" all**

Archive with source files is a part of the electronic version of this work. In step 3 *LevelDB* library is built. After successful execution of step 4 there should appear a *bin* folder with compiled PyPlag library. This library is then used in step 6 to compile and run the Java frontend. Ant property *javafx.path* should point to the location of *JavaFX* lib folder.

4.4.2 PyPlag API

To use PyPlag as a part of a larger system, the user can take advantage of the pipeline API. Using PyPlag in this manner is not necessary, but it can simplify the use of the system for simple tasks like updating the database and invoking the comparison process. The pipeline components can be also used directly via their respctive APIs. This approach would probably occur in scenarios in which the user is creating further analysis of source code units on top of PyPlag.

Pipeline API is represented as **app** object which encapsulates components. Upon initializing the object, the database is created (or opened) and the hasher component is initialized. Following methods can be called on the object:

- **add_to_db** adds subtrees of an AST represneting one source file into the database,
- **clear_db** removes all the data from the database,
- **build_tree** creates an AST from source file,
- **compare** compares input AST with ASTs stored in the database.

Figure 4.4 shows an example of API use. This code opens the database in folder *path/to/db* and initializes the hasher with tree size requirement *10*. Then it creates AST representation of source code located in file *file.py* and compares this AST to ASTs stored in the database. List of returned duplicates can then be processed further by the user.

```
app a("path/to/db", 10, app::language::python);
ast_node_ptr tree = a.build_tree("file.py");
vector<duplicate> result = a.compare(*tree);

for(auto&& d: result) {...}
```

Figure 4.4: PyPlag API example

4.4.3 PyPlag GUI

GUI frontend does exactly the same thing as the pipeline API but it can be used without creating a wrapper program around it. Parts of GUI directly reflect API methods described in the previous section. Main parts of the frontend are:

- **Database tab** 4.5 is the first tab of the main window. This part is used for building the database. Main difference between GUI and pipeline API is the fact that while using GUI the user does not need to specify the folder where the database should be built. Instead the database location is always in the same folder where the frontend executable is. Elements present in this tab are:
    1. Text area used for taking the input from the user. Folder with files to be processed and stored in the database is drag-and-dropped to the area. Alternatively URL of git repository can be written into the text area.
    2. Tree size requirement for hashing input field.
    3. After clicking this button, the input files are processed and stored in the database.
    4. After clicking this button, the database is cleared.
    5. Status bar showing the progress of the computation.

- **Compare tab** 4.6 is the second tab of the main window. It is used to invoke the comparison process. Controls present in this tab are:

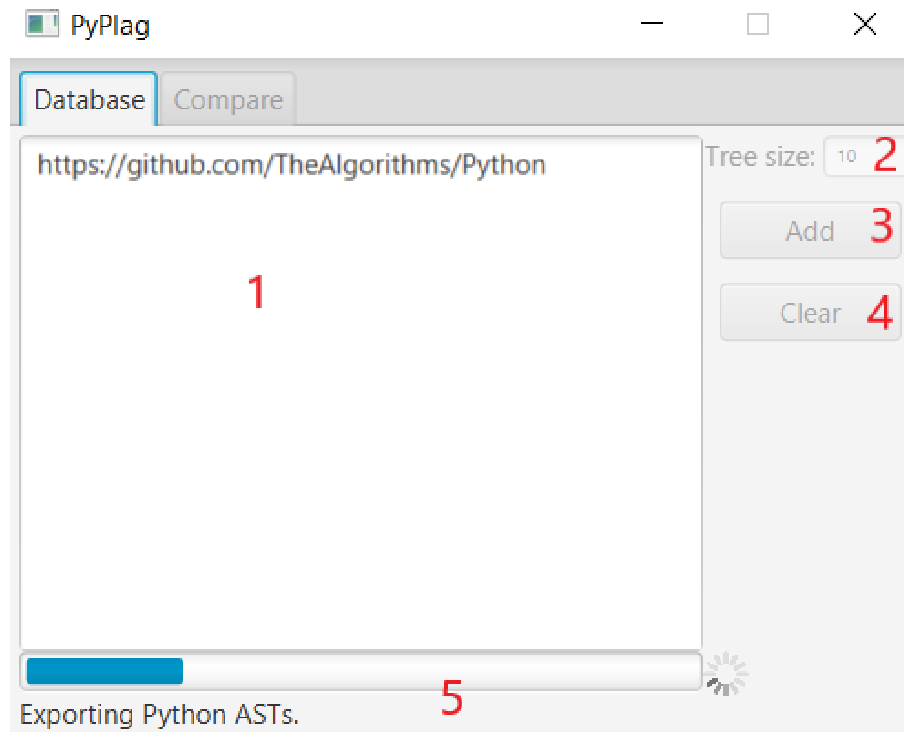

Figure 4.5: GUI database tab

1. Text are used for taking the inpur from the user. User drag-and-drops files that are to be compared with the files in the database.
2. Input field for minimal tree size parameter.
3. After clicking this button, the comparison process starts.
4. Status bar showing the progress of the computation.

- **Output window** 4.7, 4.8 showing the clones found in the input files. For each clone there is reconstruction of the source code fragment, with the list of locations of the fragment in other files. Name of the source file is highlighted with green color, code fragment location with red color and start of the clone list is marked with blue string *clones*.

Example

This is an example of PyPlag usage via PyPlag GUI. As an example input we are using some of the source files used during the experiments, described in chapter 5. Source files are part of the electronic version of this work. We will be using the *home_assignment* folder as input files and *10* as subtree hashing size.

After running the PyPlag GUI the user should see the database tab. To create the database these steps are used:

1. Drag-and-drop the *files* folder from *home_assignment* folder to the input text area (figure 4.5 1) or type-in the absolute path of the folder.

2. Type number *10* as *tree size* input (figure 4.5 2).

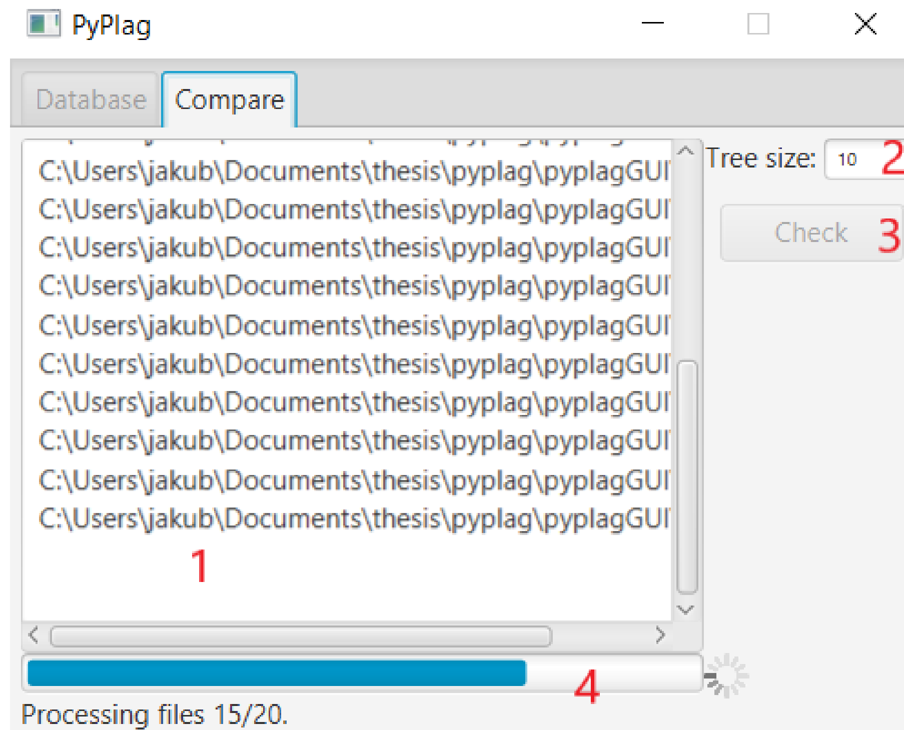

Figure 4.6: GUI compare tab

3. Click the *Add* button (figure 4.5 3).

Now the input files are being processed. The status bar (figure 4.5 5) on the bottom of the window should go from ”Exporting Python ASTs.” to ”Updating database.” and ”Cleaning up.”. After the status bar dissapears, the database has been created.

To find duplicates in a file do the following:

1. Click on *Compare* tab (figure 4.6).
2. Drag-and-drop an arbitrary file from the *files* folder into the text area or type-in the absolute path of the file (figure 4.6 1).
3. Type number *10* as *tree size* input (figure 4.6 2).
4. Click the *Check* button (figure 4.6 3).

Status bar (figure 4.6 4) on the bottom of the window now indicates the progress of computation. After the bar disappears, the output window should appear showing the duplicate code fragments that have been found in the database.

```
C:\Users\jakub\Documents\thesis\pyplag\pyplagGUI\gaus.py
    line: -1, col: -1 -> Module
        '''\nGaussian elimination method for solving a system of linear equations.\nGaussian elimination - https://en.wikipedia.org/wiki/Gaussian_elimination\n'''
        import numpy as np
        def retroactive_resolution(coefficients : np.matrix, vector : np.array) -> np.array:
            '''\n    This function performs a retroactive linear system resolution\n        for triangular matrix\n    Examples:\n        2x1 + 2x2 - 1x3 = 5        2x1 + 2x2
            (rows, columns) = np.shape(coefficients)
            x = np.zeros((rows, 1), dtype=float)
            for row in reversed(range(rows)):
                sum = 0
                for col in range(row + 1, columns):
                    sum += coefficients[(row, col)] * x[col]

                x[(row, 0)] = vector[row] - sum / coefficients[(row, row)]

            return x

        def gaussian_elimination(coefficients : np.matrix, vector : np.array) -> np.array:
            '''\n    This function performs Gaussian elimination method\n    Examples:\n        1x1 - 4x2 - 2x3 = -2        1x1 + 2x2 = 5\n        5x1 + 2x2 - 2x3 = -3
            (rows, columns) = np.shape(coefficients)
            if rows != columns:
                return []

            augmented_mat = np.concatenate((coefficients, vector), axis=1)
            augmented_mat = augmented_mat.astype("float64")
            for row in range(rows - 1):
                pivot = augmented_mat[(row, row)]
                for col in range(row + 1, columns):
                    factor = augmented_mat[(col, row)] / pivot
                    augmented_mat[col, :] -= factor * augmented_mat[row, :]


            x = retroactive_resolution(augmented_mat[:, 0:columns], augmented_mat[:, columns:columns + 1])
            return x

        if __name__ == "__main__":
            import testmod
            doctest.testmod()
```

Figure 4.7: GUI output 1

```
line: 71, col: 4 -> Assign
    x = retroactive_resolution(augmented_mat[:, 0:columns], augmented_mat[:, columns:columns + 1])

    clones:
        1.000000 gaussian_elimination.py -> line: 73, col: 4 (x=x)


line: 71, col: 8 -> Call
    retroactive_resolution(augmented_mat[:, 0:columns], augmented_mat[:, columns:columns + 1])

    clones:
        1.000000 gaussian_elimination.py -> line: 73, col: 8 (retroactive_resolution=retroactive_resolution,augmented_mat=augmented_mat,columns=columns)


line: 78, col: 0 -> If
    if __name__ == "__main__":
        import testmod
        doctest.testmod()

    clones:
        0.800000 aliquot_sum.py -> line: 43, col: 0

        0.800000 lamberts_ellipsoidal_distance.py -> line: 80, col: 0

        0.800000 haversine_distance.py -> line: 53, col: 0

        0.800000 gaussian_elimination.py -> line: 80, col: 0

        0.800000 sol1.py -> line: 39, col: 0
```

Figure 4.8: GUI output 2

5. Experiments

To verify that our method of source code clone detection works, we we carried out a series of experiments. The goal of these experiments was to determine the ability of PyPlag to identify common code fragments across multiple pairs of source files. Source files used as input data were extracted from *ReCodEx* [43], a system for dynamic analysis and evaluation of programming exercises. The system is used across majority of programming courses at Faculty of Mathematics and Physics of Charles University, therefore the data used is a real life example of working code. The fact that the source files were created by the students provides us with multiple interesting contexts in which we can analyze the data. We decided to take a closer look at code duplicity in these situations:

- Source files created by one student for one assignment.
- Source files created by different students for one assignment.
- Source files created by one student for multiple assignments.
- Source files created by different students for multiple assignments.

Based on this division of the data we came up with two different experiments:

- Recognition of files created by the same autor in the context of a single assignment. Files created by the same student for an assignment are in most cases created iteratively, by a series of corrections of mistakes made in previous versions of the solution. Therefore these files should share code fragments which were mistake-free. The number of duplicate code fragments should be significantly smaller when comparing solutions of different students. This experiment is divided into two parts. The first part focuses on exercise which was assigned as a home assignment, while the second part analyzes an exam assignment.
- Recognition of files created by the same author in the context of multiple assignments. In this experiment we focus on the ”handwriting” of a programmer. Even though the solutions of different assignments should not share the duplicate code, there may still be duplicate code fragments representing the commonly used elements of the language. Focus is on assigning the way these language elements are used in different source files to the correct author.

For both experiments we picked exercises from beginner programming courses. Source files of the solutions for these exercises have little to none teacher provided code, which could influence the results because the same chunk of the code would be present in every file. Another reason is the bigger variety of exercises and the bigger number of different students that worked on the same assignment.

In both experiments we measured to what extent are the pairs of source files similar. The similarity of the source code is represented by the formula:

$$\frac{2 \cdot c}{t_1 + t_2},$$

where c represents the number of duplicate nodes shared by AST representations of the source files, t_1 and t_2 are the numbers of nodes of the respective ASTs. Value calculated by the formula is 0 in case there are no duplicates and 1 in case the trees are identical. For easier readability of the results we multiplied the value by 100.

After the results of the comparison were calculated, we created histograms of gathered values to represent the results of the experiment. Then we used Mann-Whitney U test [44] to determine if the difference between histograms is statistically significant.

The minimal size of hashed subtree was in all experiments set to *10*. This is big enough to filter out insignificant parts of the source code, like imports of standard libraries, and at the same time small enough so no important semantic information is lost.

5.1 Home assignment

For this part of the first experiment we picked Eratosthenes sieve assignment solutions. Figure 5.1 shows histogram of similarity values for pairs of solutions created by differetn authors and figure 5.2 shows the similarity of the solutions of the same author. There is not much of an overlap between the histograms, implying that PyPlag was able to identify more subtree clones in solutions of the same author. Mann-Whitney U test confirms that the difference between two result sets is statistically significant.

Based on the results we picked the value **12** as the treshold for stating the two files were written by the same author. Table 5.1 shows numbers of true/false positives and negatives. These values yield the recall 0.67 and the precision 0.91. The average size of an AST of the final student submission was 93.18 nodes.

We noticed that in some cases students submitted source files that did not contain the implementation of Eratosthenes sieve, but rather only the demo program with constant output. This leads to smaller similarity within the group of files created by the same author and explains the relatively big number of false negatives.

	positives	**negatives**
false	21	111
true	224	424

Table 5.1: Eratosthenes sieve statistics

5.2 Exam assignment

Second part of the author recognition within the context of one assignment analyzes the submissions of students for an exam assignment. The goal was to implement the simple simulation of a car moving on the 2D grid. The two differences between the home assignment part of the experiment and the exam part are:

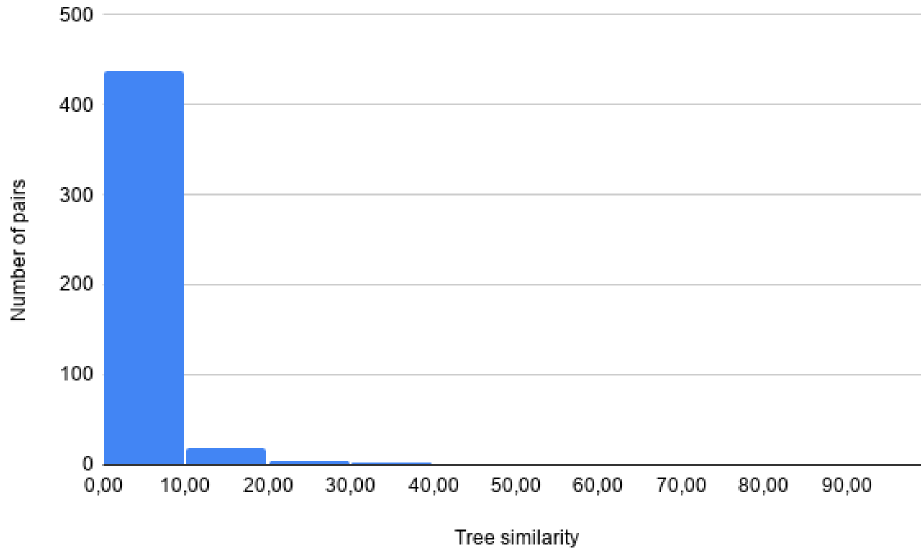

Figure 5.1: Eratosthenes sieve - different authors

- The average size of an AST of the final student submission was 294.67 nodes. That is more than three times bigger than in the first part of the experiment.
- Majority of the students submitted implementation of the assignment solution every time. There were no "demo" submissions.

Figures 5.3 and 5.4 show histograms of similarity values. Difference between result sets was statistically significant, same as in the first part of the experiment. With the number **12** as the treshold for similarity of source files, we get recall 0.84 and precission 0.99 based on the table 5.2.

While duplicates in source files of different authors have similar distribution as in the first part of the experiment, the distribution changes for the files of the same author. The first bucket contains smaller portion of the file pairs and the most pairs had similarity of 60 to 70. This shows that students modify their code less between submissions during the exam than between home assignment submissions.

	positives	**negatives**
false	2	111
true	570	188

Table 5.2: Car simulation statistics

5.3 Handwriting

In this experiment we focus on finding duplicates across source files submitted by the single author as solutions for multiple assignments. We have selected

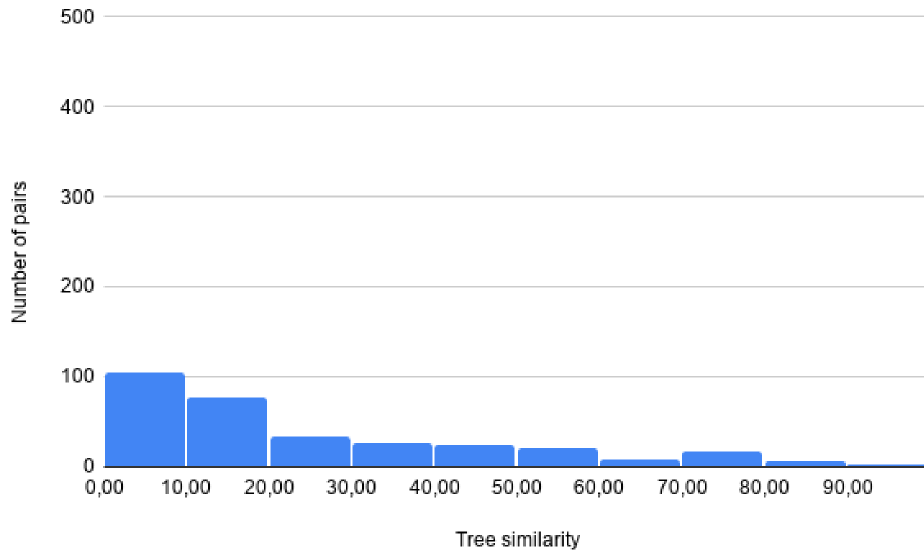

Figure 5.2: Eratosthenes sieve - same author

the set of assignments from a single programming course. Assignments cover broad spectrum of topics, so it is unlikely that potentional clones would represent reused code. Instead the duplicates should show frequently used programming techniques that the author prefers. The average size of an AST of the final student submission was 156.02 nodes.

Figures 5.5 and 5.6 show the results of the experiment. Histograms do overlap significantly, which means that PyPlag was unable to identify the files written by the single author in the context of multiple assignments. Mann-Whitney U test confirms that the difference between result sets is insignificant.

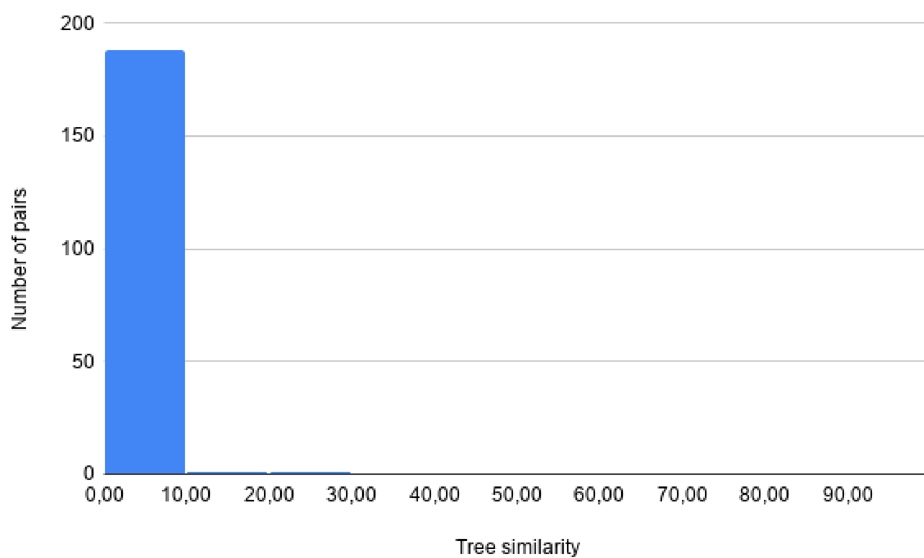

Figure 5.3: Car simulation - different authors

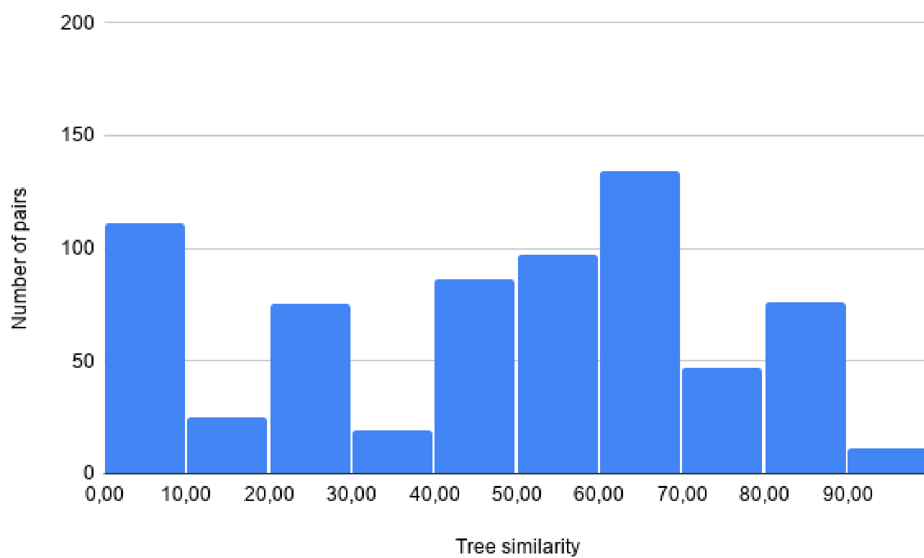

Figure 5.4: Car simulation - same author

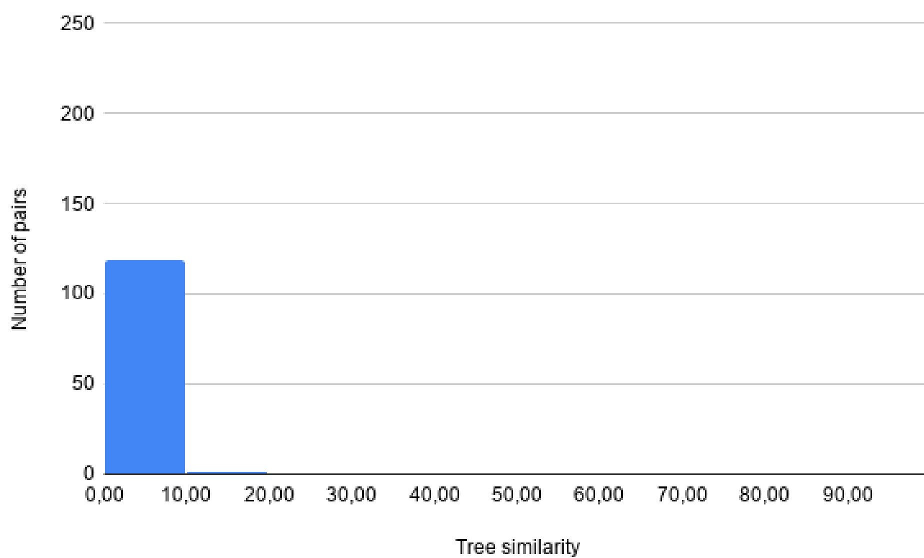

Figure 5.5: Handwriting - different authors

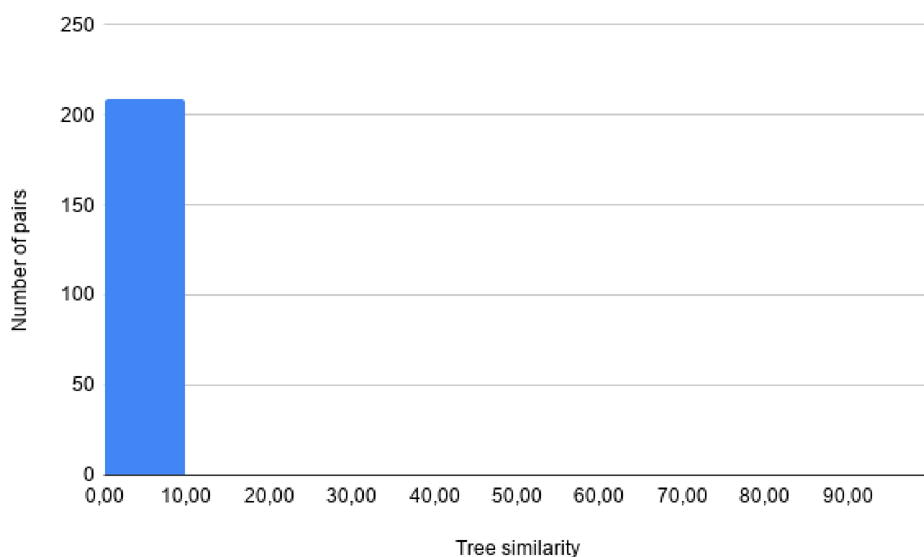

Figure 5.6: Handwriting - same author

Conclusion

The goal of this work was to design and develop a tool for syntax-driven duplicate code detection. We designed the pipeline for source code clone detection, and implemented *PyPlag*, the system that is able to detect duplicate code fragments in Python source files. The logic responsible for identifying duplicate code uses AST based method of clone detection - *fingerprinting*. This approach preprocesses the AST representation of the program, which makes the comparison of subtrees faster and less complex.

Our implementation of AST fingerprinting focuses on recognition of the trees with identical shape. The details about concrete values of identifiers, numeric and string literals are separated from the information about the tree shape. This allows us to perform the comparison of the trees in two phases. First the clone candidates are identified by comparing the tree hashes. Subsequently the identifier analysis is performed, focusing on the order of distinct literals. If the order of identifiers is the same in both trees, the clone is detected. Because this approach focuses on the structure of AST representing the code rather than on the textual representation, it is able to recognize clones created by format alternation and identifier renaming. Clones created by control replacement and code insertion change the AST structure and result in different tree hashes. This spoils the detection process. However, if large chunks of the original code are left intact, clone can be recognized by hashing the sequence of statement nodes.

PyPlag is ready to support multiple programming languages. Components of the pipeline which are responsible for clone detection are language independent and can be reused. Only language dependent parts of the pipeline (parser and semantic analyzer) need to be implemented for every new language. The way in which these components should behave is described in the chapter 4 of this work.

The pipeline can be used as a whole via common API. The components can be also used individually. This flexibility provides an easy way for extending the PyPlag, or building a larger system on top of it. In addition to API, we developed GUI for PyPlag. It simplifies the work with the system and provides the user with visualization of the results.

To verify that PyPlag is able to detect duplicate code, we performed experiments with student submissions for various assignments. The focus was on recognition of submissions created by the same author for the single assignment. Files with the common author should contain more duplicate code because they usually fix the previously made mistakes while the rest of the code stays the same. PyPlag was able to detect significantly more duplicate code fragments in the files created by the same author than in the files created by different authors, both in the home assignment and in the exam assignment setting. The best recall for an experiment was 0.84 with precision 0.99. The small number of false negatives implies that PyPlag may be used as a plagiarism detection tool as well as a software engineering tool for discovering opportunities for code refactoring.

This work has fullfilled the defined goals. The main contribution is application of existing clone detection methods to the context of Python programming language.

Future work

The obvious way to improve PyPlag is to add the support for clone detection in other programming languages besides Python. However, there are more ways how to improve the system.

Our implementation of clone detection pipeline depends on multiple third party tools and libraries. This brings more complexity to PyPlag and complicates its use. Good example of this is the use of *astexport* tool. For PyPlag to be able to create an AST of a Python source file, the source file first needs to be preprocessed by *astexport*. The result of preprocessing is JSON serialization of AST which is then parsed by the PyPlag parser. The JSON serialization of AST is significantly larger than the original Python source code. This leads to more time spent on parsing of the input. Solution of this problem would be the implementation of Python parser which would process the source files directly.

Another area that needs improvement is the GUI output. In its current form the output contains all the important information, but it is designed to be read by the user as a proof of the fact that identified clones exist. It becomes difficult to navigate with multiple input files. Solution to this would be new design of the GUI output window, which would allow for navigation and listing of identified code duplicates.

Bibliography

[1] Measure Of Software Similarity. `https://theory.stanford.edu/~aiken/moss/`. Accessed: 2020-03-20.

[2] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[3] L. Chao, C. Chen, H. Jiawei, and S.Y. Philip. GPLAG: detection of software plagiarism by program dependence graph analysis. *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 872–881, 2006.

[4] P. Gautam and H. Saini. Various Code Clone Detection Techniques and Tools: A Comprehensive Survey. *1. Critical Review on Software Testing: Security Perspective*, pages 655–667, 2016.

[5] S. Ducasse, M. Reiger, and S. Demeyer. A Language independent approach for detecting duplicated code. *Proceedings of the 1st IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[6] G.A. Di Lucca, M. Di Penta, A.R. Fasolino, and P. Granato. An approach to identify duplicated web pages. *Proceedings of the 7th IEEE Workshop on Empirical Studies on Software Maintenance*, pages 107–113, 2001.

[7] Tree edit distance. `http://tree-edit-distance.dbresearch.uni-salzburg.at/`. Accessed: 2020-03-21.

[8] I.D. Baxter, A. Yahin, L. Moura, and M.S. Anna. Clone detection using abstract syntax trees. *Proceedings of the 14th IEEE International Conference on Software Maintenance*, pages 368–377, 1998.

[9] V. Wahler, D. Seipel, G. Fischer, and M.S. Anna. Clone detection in source code by frequent item set techniques. *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 128–135, 2004.

[10] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9, 1987.

[11] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. *Proceedings of the 1st IEEE International Conference on Software Maintenance*, pages 244–254, 1996.

[12] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. *Proceedings of the 13th IEEE Working Conference on Reverse Engineering*, pages 253–262, 2006.

[13] R. Tairas and J. Gray. Phoenix-based clone detecion using suffix trees. *Proceedings of the 44th ACM Annual Southeast Regional Conference*, pages 679–684, 2006.

[14] L. Jiang, G. Mishergi, Z. Su, and S. Glondu. Scalable and accurate tree-based detection of code clones. *Proceedings of the 20th ACM Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[15] M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the 29th IEEE International Conference on Software Engineering*, pages 96–105, 2007.

[16] L. Prechelt, G. Maplpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 2002.

[17] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2003.

[18] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2003.

[19] Python ast visualizer. `https://github.com/ivan111/vpyast`. Accessed: 2020-03-25.

[20] M. Collins and N. Duffy. Convolution Kernels for Natural Language.

[21] D. Fu, Y. Xu, H. Yu, and B. Yang. WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection. *Scientific Programming*, 2017, 2017.

[22] M. Chilowicz, E. Duris, and G. Roussel. The Making of Python. *Artima*, 2003.

[23] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 1987.

[24] The MD5 Message-Digest Algorithm. `https://tools.ietf.org/html/rfc1321`. Accessed: 2020-04-14.

[25] Flex. `https://github.com/westes/flex`. Accessed: 2020-04-07.

[26] Bison. `https://www.gnu.org/software/bison/`. Accessed: 2020-04-07.

[27] The State of the Octoverse. `https://octoverse.github.com/`. Accessed: 2020-04-11.

[28] B. Venners. Syntax tree fingerprinting: a foundation for source code similarity detection. 2009.

[29] Python 2.7 Release Schedule. `https://legacy.python.org/dev/peps/pep-0373/`. Accessed: 2020-04-12.

[30] PyDatalog. https://sites.google.com/site/pydatalog/. Accessed: 2020-04-12.

[31] The Zen of Python. https://www.python.org/dev/peps/pep-0020/. Accessed: 2020-04-12.

[32] Tiobe. https://www.tiobe.com/documentation//tpci/Python.html. Accessed: 2020-04-12.

[33] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. 2000.

[34] Usage statistics of Python for websites. https://w3techs.com/technologies/details/pl-python. Accessed: 2020-04-12.

[35] Pyjs. https://github.com/pyjs/pyjs. Accessed: 2020-04-12.

[36] SQLAlchemy. https://github.com/sqlalchemy/sqlalchemy. Accessed: 2020-04-12.

[37] Python 3 type hierarchy. https://en.wikipedia.org/wiki/File:Python_3._The_standard_type_hierarchy.png. Accessed: 2020-04-12.

[38] Python ast module. https://docs.python.org/3/library/ast.html. Accessed: 2020-04-13.

[39] Astexport tool. https://github.com/fpoli/python-astexport. Accessed: 2020-04-13.

[40] LevelDB. https://github.com/CommanderBubble/MD5. Accessed: 2020-05-28.

[41] Md5 implementation. https://github.com/google/leveldb. Accessed: 2020-05-13.

[42] Pyplag. https://github.com/JakubSaksa/pyplag. Accessed: 2020-05-13.

[43] ReCodEx. https://recodex.mff.cuni.cz/. Accessed: 2020-06-09.

[44] Mann-Whitney U test. https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test. Accessed: 2020-06-09.

List of Figures

1.1 Format alternation 5
1.2 Identifier renaming 6
1.3 Statement reordering 6
1.4 Control replacement 7
1.5 Control insertion 7

2.1 AST format alternation 13
2.2 AST identifier renaming 14
2.3 AST statement reordering 15
2.4 AST control replacement 16
2.5 AST code insertion 17

3.1 Building the database 21
3.2 Clone detection 22
3.3 Leaves capturing the concrete values 25

4.1 Python 3 types [37] 32
4.2 Effect of different size limits on literal mapping 38
4.3 Communication between frontend and PyPlag 44
4.4 PyPlag API example 48
4.5 GUI database tab 49
4.6 GUI compare tab 50
4.7 GUI output 1 51
4.8 GUI output 2 51

5.1 Eratosthenes sieve - different authors 54
5.2 Eratosthenes sieve - same author 55
5.3 Car simulation - different authors 56
5.4 Car simulation - same author 56
5.5 Handwriting - different authors 57
5.6 Handwriting - same author 57

List of Tables

1.1 Approach robustness comparison [3] 10

4.1 LevelDB tables structure 39
4.2 LevelDB benchmark setup [40] 40
4.3 LevelDB benchmark [40] 40

5.1 Eratosthenes sieve statistics 53
5.2 Car simulation statistics 54

List of Abbreviations

AST Abstract Syntax Tree
BFS Breadth First Search
CFG Context Free Grammar
DRY Don't Repeat Yourself
GUI Graphical User Interface
JNI Java Native Interface
PDG Program Dependency Graph
TED Tree Edit Distance

A. Python 3.7.3 grammar

mod	Module(stmt* body, type_ignore *type_ignores) Interactive(stmt* body) Expression(expr body) FunctionType(expr* argtypes, expr returns) Suite(stmt* body)

stmt	FunctionDef(identifier name, arguments args, stmt* body, expr* decorator_list, expr? returns, string? type_comment)
	AsyncFunctionDef(identifier name, arguments args, stmt* body, expr* decorator_list, expr? returns, string? type_comment)
	ClassDef(identifier name, expr* bases, keyword* keywords, stmt* body, expr* decorator_list)
	Return(expr? value)
	Delete(expr* targets)
	Assign(expr* targets, expr value, string? type_comment)
	AugAssign(expr target, operator op, expr value)
	AnnAssign(expr target, expr annotation, expr? value, int simple)
	For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
	AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
	While(expr test, stmt* body, stmt* orelse)
	If(expr test, stmt* body, stmt* orelse)
	With(withitem* items, stmt* body, string? type_comment)
	AsyncWith(withitem* items, stmt* body, string? type_comment)
	Raise(expr? exc, expr? cause)
	Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
	Assert(expr test, expr? msg)
	Import(alias* names)
	ImportFrom(identifier? module, alias* names, int? level)
	Global(identifier* names)
	Nonlocal(identifier* names)
	Expr(expr value)
	Pass
	Break
	Continue

expr	BoolOp(boolop op, expr* values) NamedExpr(expr target, expr value) BinOp(expr left, operator op, expr right) UnaryOp(unaryop op, expr operand) Lambda(arguments args, expr body) IfExp(expr test, expr body, expr orelse) Dict(expr* keys, expr* values) Set(expr* elts)

	ListComp(expr elt, comprehension* generators) SetComp(expr elt, comprehension* generators) DictComp(expr key, expr value, comprehension* generators) GeneratorExp(expr elt, comprehension* generators) Await(expr value) Yield(expr? value) YieldFrom(expr value) Compare(expr left, cmpop* ops, expr* comparators) Call(expr func, expr* args, keyword* keywords) FormattedValue(expr value, int? conversion, expr? format_spec) JoinedStr(expr* values) Constant(constant value, string? kind) Attribute(expr value, identifier attr, expr_context ctx) Subscript(expr value, slice slice, expr_context ctx) Starred(expr value, expr_context ctx) Name(identifier id, expr_context ctx) List(expr* elts, expr_context ctx) Tuple(expr* elts, expr_context ctx)
expr_context	Load Store Del AugLoad AugStore Param
slice	Slice(expr? lower, expr? upper, expr? step) ExtSlice(slice* dims) Index(expr value)
boolop	And Or
operator	Add Sub Mult MatMult Div Mod Pow LShift RShift BitOr BitXor BitAnd FloorDiv
unaryop	Invert Not UAdd USub
cmpop	Eq NotEq

	Lt LtE Gt GtE Is IsNot In NotIn
comprehension	(expr target, expr iter, expr* ifs, int is_async)
excepthandler	ExceptHandler(expr? type, identifier? name, stmt* body)
arguments	(arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults, arg? kwarg, expr* defaults)
arg	(identifier arg, expr? annotation, string? type_comment)
keyword	(identifier? arg, expr value)
alias	(identifier name, identifier? asname)
withitem	(expr context_expr, expr? optional_vars)
type_ignore	TypeIgnore(int lineno, string tag)

B. Output of astexport

Input Python code

```python
if __name__ == "__main__":
  value = int(input())
  sieve = [0] * (value+1)
  count = 1
  for i in range(3, value+1, 2):
    if not sieve[i]:
      count += 1
      base = i*i
      while base <= value:
        sieve[base] = 1
        base += i

  print(count)
```

Output of astexport

{"ast_type" : "Module", "body" : [{"ast_type" : "If", "body" : [{"ast_type" : "Assign", "col_offset" : 2, "lineno" : 2, "targets" : [{"ast_type" : "Name", "col_offset" : 2, "ctx" : {"ast_type" : "Store"}, "id" : "value", "lineno" : 2}], "value" : {"args" : [{"args" : [], "ast_type" : "Call", "col_offset" : 14, "func" : {"ast_type" : "Name", "col_offset" : 14, "ctx" : {"ast_type" : "Load"}, "id" : "input", "lineno" : 2}, "keywords" : [], "lineno" : 2}], "ast_type" : "Call", "col_offset" : 10, "func" : {"ast_type" : "Name", "col_offset" : 10, "ctx" : {"ast_type" : "Load"}, "id" : "int", "lineno" : 2}, "keywords" : [], "lineno" : 2}}, {"ast_type" : "Assign", "col_offset" : 2, "lineno" : 3, "targets" : [{"ast_type" : "Name", "col_offset" : 2, "ctx" : {"ast_type" : "Store"}, "id" : "sieve", "lineno" : 3}], "value" : {"ast_type" : "BinOp", "col_offset" : 10, "left" : {"ast_type" : "List", "col_offset" : 10, "ctx" : {"ast_type" : "Load"}, "elts" : [{"ast_type" : "Num", "col_offset" : 11, "lineno" : 3, "n" : {"ast_type" : "int", "n" : 0, "n_str" : "0"}}], "lineno" : 3}, "lineno" : 3, "op" : {"ast_type" : "Mult"}, "right" : {"ast_type" : "BinOp", "col_offset" : 17, "left" : {"ast_type" : "Name", "col_offset" : 17, "ctx" : {"ast_type" : "Load"}, "id" : "value", "lineno" : 3}, "lineno" : 3, "op" : {"ast_type" : "Add"}, "right" : {"ast_type" : "Num", "col_offset" : 23, "lineno" : 3, "n" : {"ast_type" : "int", "n" : 1, "n_str" : "1"}}}}}, {"ast_type" : "Assign", "col_offset" : 2, "lineno" : 4, "targets" : [{"ast_type" : "Name", "col_offset" : 2, "ctx" : {"ast_type" : "Store"}, "id" : "count", "lineno" : 4}], "value" : {"ast_type" : "Num", "col_offset" : 10, "lineno" : 4, "n" : {"ast_type" : "int", "n" : 1, "n_str" : "1"}}}, {"ast_type" : "For", "body" : [{"ast_type" : "If", "body" : [{"ast_type" : "AugAssign", "col_offset" : 6, "lineno" : 7, "op" : {"ast_type" : "Add"}, "target" : {"ast_type" : "Name", "col_offset" : 6, "ctx" : {"ast_type" : "Store"}, "id" : "count", "lineno" : 7}, "value" : {"ast_type" : "Num", "col_offset" : 15, "lineno" : 7, "n" : {"ast_type" : "int", "n" : 1, "n_str" : "1"}}}, {"ast_type" : "Assign", "col_offset" : 6, "lineno" : 8, "targets" : [{"ast_type" : "Name", "col_offset" : 6, "ctx" : {"ast_type" : "Store"}, "id" :

”base”, ”lineno” : 8}], ”value” : {”ast_type” : ”BinOp”, ”col_offset” : 13, ”left” : {”ast_type” : ”Name”, ”col_offset” : 13, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”i”, ”lineno” : 8}, ”lineno” : 8, ”op” : {”ast_type” : ”Mult”}, ”right” : {”ast_type” : ”Name”, ”col_offset” : 15, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”i”, ”lineno” : 8}}}, {”ast_type” : ”While”, ”body” : [{”ast_type” : ”Assign”, ”col_offset” : 8, ”lineno” : 10, ”targets” : [{”ast_type” : ”Subscript”, ”col_offset” : 8, ”ctx” : {”ast_type” : ”Store”}, ”lineno” : 10, ”slice” : {”ast_type” : ”Index”, ”value” : {”ast_type” : ”Name”, ”col_offset” : 14, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”base”, ”lineno” : 10}}, ”value” : {”ast_type” : ”Name”, ”col_offset” : 8, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”sieve”, ”lineno” : 10}}], ”value” : {”ast_type” : ”Num”, ”col_offset” : 22, ”lineno” : 10, ”n” : {”ast_type” : ”int”, ”n” : 1, ”n_str” : ”1”}}}, {”ast_type” : ”AugAssign”, ”col_offset” : 8, ”lineno” : 11, ”op” : {”ast_type” : ”Add”}, ”target” : {”ast_type” : ”Name”, ”col_offset” : 8, ”ctx” : {”ast_type” : ”Store”}, ”id” : ”base”, ”lineno” : 11}, ”value” : {”ast_type” : ”Name”, ”col_offset” : 16, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”i”, ”lineno” : 11}}], ”col_offset” : 6, ”lineno” : 9, ”orelse” : [], ”test” : {”ast_type” : ”Compare”, ”col_offset” : 12, ”comparators” : [{”ast_type” : ”Name”, ”col_offset” : 20, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”value”, ”lineno” : 9}], ”left” : {”ast_type” : ”Name”, ”col_offset” : 12, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”base”, ”lineno” : 9}, ”lineno” : 9, ”ops” : [{”ast_type” : ”LtE”}]}}], ”col_offset” : 4, ”lineno” : 6, ”orelse” : [], ”test” : {”ast_type” : ”UnaryOp”, ”col_offset” : 7, ”lineno” : 6, ”op” : {”ast_type” : ”Not”}, ”operand” : {”ast_type” : ”Subscript”, ”col_offset” : 11, ”ctx” : {”ast_type” : ”Load”}, ”lineno” : 6, ”slice” : {”ast_type” : ”Index”, ”value” : {”ast_type” : ”Name”, ”col_offset” : 17, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”i”, ”lineno” : 6}}, ”value” : {”ast_type” : ”Name”, ”col_offset” : 11, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”sieve”, ”lineno” : 6}}}}], ”col_offset” : 2, ”iter” : {”args” : [{”ast_type” : ”Num”, ”col_offset” : 17, ”lineno” : 5, ”n” : {”ast_type” : ”int”, ”n” : 3, ”n_str” : ”3”}}, {”ast_type” : ”BinOp”, ”col_offset” : 20, ”left” : {”ast_type” : ”Name”, ”col_offset” : 20, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”value”, ”lineno” : 5}, ”lineno” : 5, ”op” : {”ast_type” : ”Add”}, ”right” : {”ast_type” : ”Num”, ”col_offset” : 26, ”lineno” : 5, ”n” : {”ast_type” : ”int”, ”n” : 1, ”n_str” : ”1”}}}, {”ast_type” : ”Num”, ”col_offset” : 29, ”lineno” : 5, ”n” : {”ast_type” : ”int”, ”n” : 2, ”n_str” : ”2”}}], ”ast_type” : ”Call”, ”col_offset” : 11, ”func” : {”ast_type” : ”Name”, ”col_offset” : 11, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”range”, ”lineno” : 5}, ”keywords” : [], ”lineno” : 5}, ”lineno” : 5, ”orelse” : [], ”target” : {”ast_type” : ”Name”, ”col_offset” : 6, ”ctx” : {”ast_type” : ”Store”}, ”id” : ”i”, ”lineno” : 5}}, {”ast_type” : ”Expr”, ”col_offset” : 2, ”lineno” : 13, ”value” : {”args” : [{”ast_type” : ”Name”, ”col_offset” : 8, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”count”, ”lineno” : 13}], ”ast_type” : ”Call”, ”col_offset” : 2, ”func” : {”ast_type” : ”Name”, ”col_offset” : 2, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”print”, ”lineno” : 13}, ”keywords” : [], ”lineno” : 13}}], ”col_offset” : 0, ”lineno” : 1, ”orelse” : [], ”test” : {”ast_type” : ”Compare”, ”col_offset” : 3, ”comparators” : [{”ast_type” : ”Str”, ”col_offset” : 15, ”lineno” : 1, ”s” : ”__main__”}], ”left” : {”ast_type” : ”Name”, ”col_offset” : 3, ”ctx” : {”ast_type” : ”Load”}, ”id” : ”__name__”, ”lineno” : 1}, ”lineno” : 1, ”ops” : [{”ast_type” : ”Eq”}]}}]}

C. Electronic attachments

Electronic attachments contain PyPlag source code, the software used by PyPlag and data used in experiments. Attached .zip archive has the following structure (only notable files are listed here):

- thesis_attachments
  - experiment_data
    - exam_assignment
    - handwriting
    - home_assignment
    - README.txt
    - results.csv
  - src
    - googletest
    - leveldb
    - pyplag
      - pyplag
      - pyplag_tests
    - pyplagGUI
      - resources
      - src
      - build.xml
    - python-astexport
    - configure
    - makefile
  - README.txt