NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Relational Representation of Python Abstract Syntax Trees

Nikolaos D. Karystinos-Avgerantonis

Supervisors: **Yannis Smaragdakis,** Professor NKUA
Iosif Lagouvardos, MSc Student NKUA

ATHENS

OCTOBER 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεσιακή Αναπαράσταση Αφηρημένων Συντακτικών Δένδρων της Python

Νικόλαος Δ. Καρυστινός-Αυγεραντώνης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Ιωσήφ Λαγουβάρδος, Μεταπτυχιακός φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2019

BSc THESIS

Relational Representation of Python Abstract Syntax Trees

Nikolaos D. Karystinos-Avgerantonis
R.N.: 1115201500064

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
Iosif Lagouvardos, MSc Student NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεσιακή Αναπαράσταση Αφηρημένων Συντακτικών Δένδρων της Python

Νικόλαος Δ. Καρυστινός-Αυγεραντώνης
Α.Μ.: 1115201500064

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης**, Καθηγητής ΕΚΠΑ
Ιωσήφ Λαγουβάρδος, Μεταπτυχιακός φοιτητής ΕΚΠΑ

ABSTRACT

The Python programming language enjoys wide use in many areas such as data science, web development, machine learning, etc. This makes Python an attractive target for various static analyses that aim to optimize the program or, more importantly, discover errors and security flaws. Overall, static analysis of industry-level programs written in almost any language is deemed necessary as a means of improving and maintaining the quality of increasingly sophisticated and complex software.

Fact generation is the representation of a program as a database of facts, according to a pre-defined schema (relational representation). It is an important first step to many static analyses, especially in declarative form, such as those supported by the Doop static analysis framework, which utilizes the Datalog language and a fact database to perform its analyses. The database, after its creation, allows the specification of a static program analysis as a set of (possibly recursive) queries.

In this thesis, we provide a means of transforming the AST of a Python program into an equivalent set of fact files that constitute a fact database. This provides a foundation framework, on which more complex fact generation or transformation logic can be developed. The work presented, hopefully provides a basis on which various static analyses or program transformations could be constructed for Python programs, at the AST level.

SUBJECT AREA: Static Program Analysis and Program Transformation

KEYWORDS: static program analysis, program transformation, doop framework, python, fact generation

ΠΕΡΙΛΗΨΗ

Η γλώσσα προγραμματισμού Python, χρησιμοποιείται ευρέως σε πολλούς τομείς όπως επιστήμη δεδομένων, ανάπτυξη εφαρμογών διαδικτύου, μηχανική μάθηση κ.α. Αυτό, κάνει την Python ελκυστικό στόχο για διάφορες στατικές αναλύσεις οι οποίες στοχεύουν να βελτιστοποιήσουν το πρόγραμμα ή, ακόμα σημαντικότερα, να ανακαλύψουν λάθη και κενά ασφαλείας. Γενικά, η στατική ανάλυση βιομηχανικών εφαρμογών γραμμένων σε οποιαδήποτε γλώσσα κρίνεται απαραίτητη ως μέσο βελτίωσης και διατήρησης της ποιότητας όλο και πιο προηγμένου και πολύπλοκου λογισμικού.

Παραγωγή γεγονότων είναι η αναπαράσταση ενός προγράμματος ως μια βάση δεδομένων που περιέχει γεγονότα, σύμφωνα με κάποιο ορισμένο εκ των προτέρων σχήμα (σχεσιακή αναπαράσταση). Η παραγωγή γεγονότων, είναι ένα σημαντικό πρώτο βήμα σε πολλές στατικές αναλύσεις, και ιδιαίτερα σε δηλωτικές στατικές αναλύσεις, όπως αυτές που υποστηρίζονται από το Doop framework για στατική ανάλυση, το οποίο αξιοποιεί την γλώσσα Datalog και βάσεις γεγονότων για να εκτελέσει τις αναλύσεις του. Η βάση γεγονότων, μετά την δημιουργία της, επιτρέπει τον προσδιορισμό μιας στατικής ανάλυσης προγράμματος ως ένα σύνολο από (πιθανώς αναδρομικές) επερωτήσεις.

Σε αυτή την πτυχιακή εργασία, δίνουμε ένα τρόπο μετατροπής του AST ενός προγράμματος Python σε ένα ισοδύναμο σύνολο από αρχεία γεγονότων τα οποία αποτελούν μια βάση γεγονότων. Αυτό ταυτόχρονα θεμελιώνει ένα βασικό framework, πάνω στο οποίο μπορούν να αναπτυχθούν πιο περίπλοκες παραγωγές γεγονότων ή μετασχηματισμοί. Η δουλειά που παρουσιάζεται, ελπίζουμε να αποτελέσει βάση πάνω στην οποία θα υλοποιηθούν διάφορες στατικές αναλύσεις ή μετασχηματισμοί προγραμμάτων Python, στο επίπεδο του AST.

ΠΕΡΙΟΧΗ ΑΝΤΙΚΕΙΜΕΝΟΥ: Στατική ανάλυση προγραμμάτων και μετασχηματισμός προγραμμάτων

ΛΕΞΕΙΣ-ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, μετασχηματισμός προγραμμάτων, doop framework, python, παραγωγή γεγονότων

To my mother Xanthi, my father Dimitris,
and my sister Anastasia.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and thanks to my supervisors, professor Yannis Smaragdakis and MSc student Iosif Lagouvardos, for providing the topic of this thesis and giving me the chance to work on a really interesting project.

A second round of applause should be given to Iosif, who tirelessly provided the necessary guidance and help I needed at many points, which was crucial to the completion of this thesis.

October 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1. INTRODUCTION

This thesis aims to present a way through which the structure and contents of a Python program's AST (Abstract Syntax Tree) are imported into a Java program that can then perform operations on the tree. We primarily focus on generating fact files from the AST, and more specifically, we implement the simplest possible case of fact generation : a direct translation of the AST structure into fact files. However, the framework we have constructed allows us to describe arbitrary operations on the AST using the visitor pattern, so that more complex fact generation or program transformation logic can be implemented as an extension to this work.

Python, as we will discuss later on, exposes its programs' abstract syntax trees. Our work leverages that fact to extract the AST structure in the form of a JSON file, as a first step. We then move on to import the JSON representation of the AST in our Java program. The Java code, in turn, parses the JSON file and constructs an object hierarchy practically indistinguishable from the original Python AST. The class hierarchy defined in our Java program is augmented with the visitor pattern, which allows us to describe algorithms on the tree. Finally, we construct a specialized visitor that outputs the relational structure of the AST into fact files that comprise our final fact database.

The rest of the thesis is organized as follows:

1. In chapter 2, we provide a basic overview of the Python programming language and abstract syntax trees, the software tools that were used, the visitor pattern which greatly simplifies and modularizes our implementation, and the final output of our program, fact files.
2. In chapter 3, we present - step by step - how we imported the AST in our Java program.
3. In chapter 4, we explain how the visitor pattern allows us to perform fact generation on the AST.
4. In chapter 5, we present a simple but complete example of the transformation and fact generation process.
5. In chapter 6, we give some additional insights and reflections on our work.
6. In chapter 7, some final thoughts are given, and the thesis concludes.

2. BACKGROUND

2.1 Python

Python is a high-level, interpreted, multi-paradigm programming language. It is dynamically typed and garbage collected with a design and syntax that emphasizes code simplicity and readability. Programming paradigms supported include : object-oriented programming (main approach), procedural programming, functional programming and many more. Python's design philosophy makes it an ideal choice for Rapid Application Development since the edit-test-debug cycle is insanely fast, due to the absence of the compilation step and the language's built-in exception mechanism.

Python is mainly used in data science and web development applications, along with incredibly popular frameworks like Django and Flask for web development and NumPy for data science. It is also a potent scripting or glue language, used to easily connect existing software components together. Some stats about Python are available here [4].

Overall, Python consists a powerful programming language with a comprehensive standard library and an array of useful frameworks built around it. Its wide use makes Python a target for static analysis. Some of its features however, make statically analyzing a Python program challenging (e.g. dynamic typing).

2.2 Python ASTs

An AST (Abstract Syntax Tree) consists a tree representation of the syntactic structure of a program (source code). The tree is made up of nodes, with each node representing a construct in the original source code. Constructing the AST of a program, constitutes a crucial step of the compilation process for compiled languages. The compiler uses the AST during various phases of the compilation like semantic analysis, symbol table construction and intermediate code generation.

We mentioned earlier that Python is an interpreted language. That is not entirely accurate. What in fact happens in Python's standard reference implementation, namely CPython, is a mixture of compilation and interpretation. Python source files (.py) are first compiled to byte code (.pyc). The produced bytecode is then interpreted. The aforementioned aren't true for every Python implementation, so, we could say that Python doesn't bother about being compiled or interpreted, the implementation of the language makes that decision.

With the above in mind, the notion of an AST applies to Python, as it does to most any other programming language specified by a grammar to avoid syntactic ambiguity. A valid AST is an instance of the language's grammar, a representation of a sentence (program) that follows syntactic rules (grammar) and is thus correct (syntactically) and unambiguous.

Among its countless features and modules, Python can also programmatically expose a source file's AST through the - conveniently named - **ast** module [5]. This is very crucial, practically enabling the work described in this thesis, since our fact generation is performed on the AST of the target program. Alternative approaches also exist. For example, one could perform fact generation at a lower level; in Python's case, the bytecode level.

The official documentation for Python ASTs is not comprehensive. However, there exists extremely useful and thorough unofficial documentation here [3].

2.3 Software Tools

In this section we give a brief description of the software tools utilized in this project. More details of the exact usage of the below tools can be found in the following sections [3, 4].

2.3.1 astexport

astexport is a library implemented in Python. The source code is available here [6], but astexport is also available to install through the **pip** package manager for Python. This library, allows us to effortlessly export a Python source file's AST in JSON format. In Linux, we can do so just by issuing the below command to our terminal, after installing astexport.

Listing 1: astexport command

```
astexport < input.py > ast.json
```

2.3.2 Gson

Gson [7] is an open source Java library, developed by Google, that enables the conversion of JSON strings to equivalent Java objects, but can also do the reverse operation, which is converting a Java object to its JSON representation. Briefly stated, Gson is a Java serialization/deserialization library that converts Java objects to JSON and back.

2.4 Visitor Pattern

The visitor design pattern, in the context of object-oriented programming, allows the programmer to separate algorithm and object structures. Essentially, when utilizing the visitor pattern we can add new operations to existing object structures without needing to modify the structures. As a result, instead of having to change all of our class definitions to define a new operation, we can simply specify a new visitor that defines the operation for every object.

This is accomplished by implementing what is known as a double dispatch mechanism. Most object-oriented languages support - by design - single dispatch, which means that the implementation of a function that is chosen to run only depends on the dynamic type of the receiver of the call, but not on the dynamic type of the arguments provided. Single dispatch can easily be leveraged to implement double dispatch, and that is how the visitor pattern can be implemented in single dispatch languages. What really happens, in short, is the following : We call a method on our object - let us name it *accept* (visitor pattern terminology) - which, through overriding resolves our object's dynamic type. We are now inside the implementation of *accept* in the correct class definition. One could argue this is enough, but remember that we want to be able to support the addition of different operations without modifying the class definition. Following that thought, forbids us from adding another method (e.g. *accept2*) to our class to support a second "behaviour" we want to achieve. What we do instead, is pass a visitor object (that implements our desired behaviour/algorithm on the objects) during the first call. Now, being inside the correct accept method with the help of overriding (single dispatch), we call the visitor's visit method passing in a self reference (the *this* pointer in some languages) as an argument, essentially giving the object to the visitor. The dynamic type of our visitor is resolved with single dispatch as expected so we have now "hopped out" of the class we did not want to modify, and we basically achieved our goal since we transferred the flow of execution to the implementation of a method in our desired visitor, taking into account the dynamic types of both our initial object and the visitor, thus realizing double dispatch.

The visitor pattern has obvious benefits but also drawbacks. The fact that we only have to give an accept method definition for our classes and can then perform arbitrary operations by implementing visitors is really powerful, but say, for example at some point we need to extend our class hierarchy. In that case, it would not simply suffice to provide an accept method definition for our new classes. We would have to add the corresponding visit method to all of our - already implemented - visitors, which is maybe a small price to pay considering the separation between classes and algorithms we achieved.

The above description does not really provide great intuition to someone who has not encountered the visitor pattern before, but its implementation and benefits in this work are discussed again later on, with examples, in chapter [4]. More info on the visitor pattern can be found here [8].

2.5 Facts & Fact Files

The notion of facts is of great importance in logic and logic programming. A logic program, simply expresses facts and rules that are true in a domain. The initial facts along with the application of the declared rules can produce new, previously unknown facts through ”logical deduction”. Notice how this description of logic programming fits perfectly in the context of static analysis : We have some initial facts (our program's source code or its AST), and we would like to produce output facts - that are initially unknown - regarding the behaviour of our program in all possible executions.

Doop [1] [2], exploiting the above similarity, expresses its analyses declaratively in the Datalog language, defining rules that use input facts (the Extensional Database - EDB) to produce output facts (the Intensional Database - IDB). This has an array of benefits, including simpler and more concise declarative implementations, which, in turn allow easier correctness verification, compared to a procedural implementation that sometimes might be impossible to grasp and formally verify, especially for complex analyses.

In the scope of this thesis, we consider facts as tuples, that are grouped in a file (of csv format) which we will call ***fact file***. Our fact files consist a ***fact database***. An example of a fact file could be the following in our context : Say our file is called ***NodePosition.facts***. The tuples (or facts) contained in this file could be of the form ***(NodeID, lineno, colno)***. What these facts represent, is, that every node of the AST (uniquely identified by its node ID) corresponds to a specific location in the source file. ***lineno*** gives us the line and ***colno*** the column in the original source file.

Moreover, since we will be outputting an almost direct translation of the AST into fact files, we will use (at least) one fact file per AST node type. Since, as mentioned earlier, there exist over 100 node types, our output database will contain around 100 fact files. That sounds troublesome, but not all files (or nodes) are common, so most files contain facts that are of little to no importance in a real Python program, or are completely empty. The above statement is partially true however, since we have chosen not to generate fact files for few node types as we will explain later on in our example [5].
More about this is presented in chapter [4].

3. TRANSFORMATION STEPS

3.1 Introduction

We can now begin describing how we have gone about importing the AST structure into our Java program. Note that this is not the only way this could have been done, however it was chosen since it is conveniently aided by our software tools mentioned earlier.

3.2 Step 0 : Python source code → Python AST

Thankfully, we do not have to do anything in this step. Python, as we already discussed, exposes a program's AST through the **ast** module. In the event that we were working on a language that does not expose the AST, this step would have been significantly harder. We would have had to build a parser that identifies our language's grammar, and then leverage our parser to construct the AST. Building a parser for a complex language like Python is not a simple task.

3.3 Step 1 : Python AST → JSON

This step is also relatively trivial. As we discussed in chapter [2.3], *astexport* can do all the work for us. It utilizes the *ast* module to traverse the whole AST and export a JSON representation. Let us now give an example to see how the output of *astexport* looks.

Consider the following simple Python program stored in the file ***f.py*** :

Listing 2: Simple Python function

```python
def f(x):
  return x + 1
```

This code looks dead simple, right? We just defined a function that adds one to its argument and returns the result. Now let's take a look at the JSON that represents the AST, produced by astexport. We first execute this command :

Listing 3: astexport command for pretty JSON

```
astexport -p < f.py
```

Optional argument -p gives us a prettified JSON output that looks something like this:

Listing 4: Original JSON for f.py

```json
{
    "ast_type": "Module",
    "body": [
        {
            "args": {
                "args": [
                    {
                        "annotation": null,
                        "arg": "x",
                        "ast_type": "arg",
                        "col_offset": 6,
                        "lineno": 1
                    }
                ],
                "ast_type": "arguments",
                "defaults": [],
                "kw_defaults": [],
                "kwarg": null,
                "kwonlyargs": [],
                "vararg": null
            },
            "ast_type": "FunctionDef",
            "body": [
                {
                    "ast_type": "Return",
                    "col_offset": 1,
                    "lineno": 2,
                    "value": {
                        "ast_type": "BinOp",
                        "col_offset": 8,
                        "left": {
                            "ast_type": "Name",
                            "col_offset": 8,
                            "ctx": {
                                "ast_type": "Load"
                            },
                            "id": "x",
                            "lineno": 2
                        },
                        "lineno": 2,
                        "op": {
                            "ast_type": "Add"
                        },
                        "right": {
                            "ast_type": "Num",
                            "col_offset": 12,
                            "lineno": 2,
                            "n": {
                                "ast_type": "int",
                                "n": 1,
                                "n_str": "1"
                            }
                        }
                    }
                }
            ],
            "col_offset": 0,
            "decorator_list": [],
            "lineno": 1,
            "name": "f",
            "returns": null
        }
    ]
}
```

Wait, what happened? 2 lines of Python correspond to 60 lines of JSON for the AST? That seems absurd, however this is what in fact happens. Most of the AST nodes have many fields, most of which are usually unused. For example the node for function definitions contains subnodes for storing annotations (which we didn't use) or decorators (which again we didn't use). Let's simplify the above JSON to only include the necessary - for our example - fields. We also remove fields with key *lineno* and *col_offset* which correspond to the line and column (in the source file) our node was derived from.

Our new simplified version looks something like this :

Listing 5: Simplified JSON for f.py

```
{
    "ast_type": "Module",
    "body": [
        {

            "ast_type": "FunctionDef",
            "name": "f",
            "args": {
                "ast_type": "arguments",
                "args": [
                    {
                        "arg": "x",
                        "ast_type": "arg"
                    }
                ]
            },
            "body": [
                {
                    "ast_type": "Return",
                    "value": {
                        "ast_type": "BinOp",
                        "left": {
                            "ast_type": "Name",
                            "id": "x",
                            "ctx": {
                                "ast_type": "Load"
                            }
                        },
                        "op": {
                            "ast_type": "Add"
                        },
                        "right": {
                            "ast_type": "Num",
                            "n": {
                                "ast_type": "int",
                                "n_str": 1,
                            }
                        }
                    }
                }
            ]
        }
    ]
}
```

Let's point out some things to clarify what we are seeing:

- Each object in our JSON enclosed in braces { } corresponds to a node of our AST.
- Some nodes contain fields whose values are not single-child nodes or scalars (atomic values), but instead many nodes grouped in an array of JSON objects. For example, the *body* field of the outermost JSON object contains an array of nodes. In our case the array has only one object, but, generally, that field corresponds to the body of our program, and most programs do not simply contain a function definition and nothing more.
- All objects have a field named *ast_type*. This field signifies the type of node the object represents. Some of the node types we notice are : *FunctionDef, Return, BinOp, Add, Num*, which makes sense considering what our program does.

Take a look at [line 4] of our simplified JSON where a new node begins. The node is a function definition, the function name is *f*, there are arguments and a body that contains a return node. Hopefully, by now you can see the correlation between the AST structure and our original program.

We will not go into more details right now, but the JSON representation of the AST will be revisited in our example [5].

3.4 Step 2 : JSON $\rightarrow$ Java objects

Having acquired the JSON representation of the AST we now have to convert the structure described by the JSON file to Java objects. This step is slightly more complex, but, once again, our tools, and more specifically [Gson], greatly simplify what we have to do. Had we not used Gson, we would have had to build a JSON parser ourselves, and then leverage the parser to instantiate the proper Java objects that correspond to the JSON objects in our file. Gson helps us automatically parse the JSON file, provided we specify the object structures that will be encountered during the parsing.

Essentially, we have to define a Java class for every type of node in the AST, with fields that exactly match those encountered in the JSON object of that type. There is a small but very important detail however : JSON objects do not inherently carry type information. The node type is given by the *ast_type* field. On the contrary, Java objects inherently carry their type, without us having to define it as an attribute. So, in Java, as in most object-oriented languages, the type of an object coincides with the name of the class the object is an instance of.

Abstractly, what we have to do is this : Tell Gson to parse the JSON file, and, for every JSON object encountered, instantiate a Java object of the class with name that matches the value of the *ast_type* field of the JSON object, and then copy the rest of the JSON values to the corresponding attributes of our new Java object. In this way, we will have a Java object of the correct type that has been fully instantiated.

Let us elaborate on how we construct our class hierarchy and use Gson to parse our JSON file.

We first define the Node class as an abstract base class.

Listing 6: Abstract Node Class Definition

```java
public abstract class Node {}
```

We will now define, for the sake of simplicity, just the four derived classes that help represent numbers. The rest of the classes are defined in the same manner, one for every node type, with appropriate fields. All node descriptions can be found here [3].

Listing 7: Num Class Definition

```java
public class Num extends Node {
    public int lineno;
    public int col_offset;
    public Node n;
}
```

Listing 8: Int Class Definition

```java
public class Int extends Node {
    public long n;
    public String n_str;
}
```

Listing 9: Float Class Definition

```java
public class Float extends Node {
    public double n;
}
```

Listing 10: Complex Class Definition

```java
public class Complex extends Node {
    public double i;
    public double n;
}
```

Notice that:

- The ***Num*** node contains a subnode ***n***. A **Num** node can have an ***Int, Complex*** or ***Float*** subnode, so the dynamic type of ***n*** could be any of those, and that is one of the reasons why we need our base class. Different types of subnodes could possibly appear at the same field of the base node.
- The ***Num*** node contains ”position” fields ***lineno*** and ***col_offset*** while its possible subnodes do not.
- The ***Int*** node also contains a string representation of the integer to handle Python’s arbitrary integer size.

We will also give the object structure of a control-flow node, the ***While*** node.

Listing 11: While Class Definition

```java
public class While extends Node {
   public int lineno;
   public int col_offset;

   public Node test;
   public Node[] body;
   public Node[] orelse;
}
```

Listing 12: While Structure Pseudocode

```
while([test]):
    [body]
else:
    [orelse]
```

Two remarks for the previous two listings:

- The ***body*** and ***orelse*** fields are of type ***Node[ ]*** since they possibly hold many nodes. For example if our ***while*** contained two assignments on separate lines, the ***body*** array would contain two ***Assign*** nodes. Similarly for the ***orelse*** array.
- We again have positional information for our ***While*** node.

As a next step we have to declare our base class to Gson and give it the field that defines the subtype we want to instantiate. Our next code example will illustrate how this is done for all our classes using reflection. Note that we have grouped the node classes in larger superclasses (outer classes) that essentially constitute node categories. The code below is a snippet residing in our main function. Some variables may not be properly declared in the snippet.

Listing 13: Initializing and deploying the Gson parser

```java
Class[] nodeCategories = { Literals.class, Variables.class, Expressions.class,
                           Statements.class,  ControlFlow.class,
                           Definitions.class, AsyncAwait.class, Misc.class };

Map<Class<?>, String> classLabels = new HashMap<Class<?>, String>();

/* General Case : Label = Class name */
for (Class nodeCategory : nodeCategories) {
    for (Class clazz : nodeCategory.getDeclaredClasses()) {
        classLabels.put(clazz, clazz.getSimpleName());
    }
}

/* Special Case : Custom Labels */
classLabels.put(Literals.Int.class, "int");
classLabels.put(Literals.Float.class, "float");
classLabels.put(Literals.Complex.class, "complex");

RuntimeTypeAdapterFactory<Node> nodeAdapterFactory =
                 RuntimeTypeAdapterFactory.of(Node.class, "ast_type");

for (Class clazz : classLabels.keySet()) {
    nodeAdapterFactory.registerSubtype(clazz, classLabels.get(clazz));
}

Gson gson = new GsonBuilder()
        .serializeNulls()
        .registerTypeAdapterFactory(nodeAdapterFactory)
        .create();

Node root = gson.fromJson(bufferedReader, Node.class);
```

Let's now point out what's happening in the above snippet:

- We first define a Class object array to hold our superclasses (outer classes).
- Then, we define the map ***classLabels*** that helps us correlate class objects with the class labels used by ***astexport*** (we use the term ***class labels*** to talk about the possible contents of the ***ast_type*** field described earlier).

- For every category (outer class), we use the reflective method ***getDeclaredClasses*** to obtain all the node types defined within that category as Class objects, and insert them in the map with their names as labels. We then manually set some labels, due to ***int*** and ***float*** being keywords in Java. We can't define a class named ***int*** but ***astexport*** sets the field ***ast_type*** to ”int”, so our map must correlate the ***Int*** class object with the string ”int” that will be encountered during the JSON parsing. We do the same for ***Float*** due to similar reasons and to ***Complex*** for symmetry since these are all the numeric nodes. The rest of the classes' labels coincide with their names.
- After this, we construct a ***RuntimeTypeAdapterFactory*** object, passing in our base class and the name of the JSON field that will contain the class label. The runtime type adapter factory does the hard work of dynamically reading the field ***ast_type*** for every JSON object and instantiating an object of the class that corresponds to the class label it encountered. For this to happen, we also have to register our classes and their labels, so we simply use our map ***classLabels*** to do that (line 22).
- Having done the above, we can now instantiate a parser object, also registering our ***RuntimeTypeAdapterFactory*** object as part of the parser.
- Finally, all we have to do is give our parser a reader on the file and, magically, we get back the root of a fully materialized AST that mirrors the description in our JSON file.

We have now completed the transformation and importing process. The whole AST resides in our Java program's memory ready to be worked on.

3.5 Transformation Summary

Overview of what we have done in this section:

- Show how ***astexport*** is used to produce the JSON representation of the AST from our initial Python source file.
- Outline the definition of our class hierarchy that corresponds to the AST's node types, presenting definitions for some selected classes.
- Elaborate on how we initialize our parser to perform the required deserialization.

4. FACT GENERATION

4.1 Overview

In this section, after having imported the AST in our Java program, we move on to implement the visitor pattern for our node hierarchy, and, additionally, present a simple fact generation visitor that allows us to represent the AST in the form of ***fact files*** as described earlier in chapter [2.5].

4.2 Visitor Pattern

The visitor pattern was briefly touched upon in chapter [2.4]. In this chapter, we demonstrate our visitor implementation, and the extensions needed in our class hierarchy to enable the visitor pattern.

The visitor pattern is especially useful on trees, and since it should be extensible we ought to be able to support different traversals of the tree, in addition to different behaviours for a traversal. More generally, any algorithm on the tree should be feasible to implement. Building a visitor hierarchy is a good design practice since a generic traversal visitor could be subtyped by a concrete visitor that only performs specialized operations on some of the nodes. In essence, we get the traversal behaviour from our base class, and our subtype simply overrides the visit methods of the nodes it wants to process.

The first step taken was building the generic visitor interface.

Listing 14: Generic Visitor Interface

```java
package visitors;

import nodes.*;

public interface NodeVisitor<R, A> {
    R visit(Literals.Num node, A arg);
    R visit(Literals.Int node, A arg);
    R visit(Literals.Float node, A arg);
    R visit(Literals.Complex node, A arg);
    . . .
    . . .
    . . .
    R visit(Misc.Module node, A arg);
}
```

Every node visitor (or its supertypes) should implement the ***NodeVisitor*** generic interface, so that it is made certain that a behaviour is implemented for every node type. The generics used correspond to the return type of the visit functions (***R***) and the argument which they might utilize (***A***).

The next logical step is to implement generic traversal visitors that will allow us to traverse the tree in various ways. We only present a depth-first traversal visitor but many others could be implemented.

Listing 15: Generic Depth-First Visitor

```java
package visitors;

import nodes.*;

public class DepthFirstVisitor<R, A> implements NodeVisitor<R, A> {

    public R visit(Literals.Num node, A arg) {
        return node.n.accept(this, arg);
    }

    public R visit(Literals.Int node, A arg) {
        return null;
    }

    public R visit(Literals.Float node, A arg) {
        return null;
    }

    public R visit(Literals.Complex node, A arg) {
        return null;
    }

    . . .

    public R visit(ControlFlow.While node, A arg) {
        node.test.accept(this, arg);
        for (Node n: node.body)
            n.accept(this, arg);
        for (Node n: node.orelse)
            n.accept(this, arg);
        return null;
    }

    . . .

    public R visit(Definitions.ClassDef node, A arg) {
        for (Node n: node.bases)
            n.accept(this, arg);
        for (Node n: node.keywords)
            n.accept(this, arg);
        for (Node n: node.body)
            n.accept(this, arg);
        for (Node n: node.decorator_list)
            n.accept(this, arg);
        return null;
    }

    . . .
}
```

Some comments on the above code snippet:

- Obviously, we don't present all the methods, as with our interface earlier, since we have around 100 methods for our different node categories. The methods however follow a simple pattern of definition, so, from our example, the remaining method definitions can easily be inferred, by consulting the node documentation [3].
- The general strategy we follow to implement the DFS traversal is this :
  - If we arrive at a leaf node (a node that has no subnodes), we simply return.
  - If our node has subnodes, or arrays of subnodes, we call the ***accept*** method on all subnodes. The ***accept*** method is the addition we need to make to our classes so that we can implement the visitor pattern.
- The accept method call takes us to the accept method body of the class that corresponds to the dynamic type of our object (single dispatch as discussed in [2.4]), and inside the body of that method, the visit method is called, which sends us back to our visitor, and so on and so forth.

The modifications to our node classes are given below. We first need to modify our Node abstract base class.

Listing 16: Original Node Abstract Class

```java
package nodes;

public abstract class Node {}
```

Listing 17: Modified Node Abstract Class

```java
package nodes;
import visitors.NodeVisitor;

public abstract class Node {
    public <R, A> R accept(NodeVisitor<R, A> v, A arg) { return null; }
}
```

The only addition we need to perform is the ***accept*** method with a body that simply returns. We then override this method in all our node subclasses in the same way presented below. This allows us to handle nodes polymorphically, while still getting us to the correct class when we call accept (which we need for the visitor pattern). The visitor pattern would not be possible to implement if we did not override the base class' ***accept*** method, because we would not be able to leverage the double dispatch mechanism we described earlier in [2.4].

We only give the modification for one node class, since the modification is identical in all other classes.

Listing 18: Original Num Node Class

```java
public class Num extends Node {
    public int lineno;
    public int col_offset;
    public Node n;
}
```

Listing 19: Modified Num Node Class

```java
public class Num extends Node {
    public int lineno;
    public int col_offset;
    public Node n;

    @Override
    public <R, A> R accept(NodeVisitor<R, A> v, A arg) {
        return v.visit(this, arg);
    }
}
```

All the ***accept*** method has to do, is call the visitor's visit method (which will resolve its dynamic type) and also pass in the ***this*** self-reference.

With all the above in mind let us look at a simple example of visiting a Num node.

Listing 20: Simple Visit Example

```java
    NodeVisitor<String, String> v = new DepthFirstVisitor<>();

    Literals.Num NumNode = new Literals.Num();
    Node n = NumNode;

    n.accept(v, null);
```

Let's mentally trace the calls:

- ***n.accept(v, null)*** seems to be calling the abstract base class' accept method but remember that we have overriden it! So the method actually called in the first step is the accept method inside the ***Num*** class that we defined in the previous listing [19].
- That method in turn, calls the visit method of our visitor. The dynamic type of our visitor is ***DepthFirstVisitor*** so the function with signature ***public R visit(Literals.Num node, A arg)*** inside the [***DepthFirstVisitor*** class] will be called.
- That function in turn calls the accept function on its subnode. This will transfer the flow of execution to the appropriate overriding accept method inside either the ***Int***, ***Float*** or ***Complex*** class, since as we mentioned, those are the possible subnodes of a ***Num*** node.
- Then the visit method is called inside that accept method, and so on, and so forth.

In general we get the pattern : $accept \rightarrow visit \rightarrow accept \rightarrow visit \ldots$

Hopefully, by now, the inner workings of the visitor pattern are slightly clearer.

Note: Our previous example would not actually compile since our node classes are enclosed in the category classes. This means, in our example, that we would need a ***Literals*** object to instantiate a ***Num*** object. This can easily be solved by making our inner classes static, but we purposefully avoided that in our previous listings for simplicity. The actual implementation declares the inner classes as static to solve this.

4.3 Fact Generation Visitor

So far, we have presented the generic visitor pattern framework that is implemented. In this section, we will elaborate on how we constructed the visitor that does the ***"simple fact generation"***, by outputting well-formed facts to appropriate fact files.

Let's take a look at part of the implementation.

Listing 21: Fact Generation Visitor

```java
package visitors;

import nodes.*;
import util.ASTRepresentation;
import util.Database;
import util.PredicateFile;
import util.Session;

import java.io.File;
import java.io.IOException;
import java.util.Stack;

public class FactGenVisitor extends DepthFirstVisitor<String, Integer> {

    String filename;

    Database db;
    ASTRepresentation _rep;
    Session sess;

    Stack<Node> scope;

    private static final String NULL_STR = "-";

    public FactGenVisitor(String filename){
        try {
            db = new Database(new File("./output"));
        } catch (IOException ex) {
            throw new RuntimeException("Could not instantiate file database.");
        }
        _rep = ASTRepresentation.getRepresentation();
        sess = new Session();
        this.filename = filename;
        scope = new Stack<>();
    }


    public void cleanup() {
        try {
            db.close();
        } catch (IOException ex) {
            throw new RuntimeException("Could not close file database.");
        }
    }

    @Override
    public String visit(Literals.Num node, Integer arg) {
        String ID = _rep.getScopedID(node, scope.peek(), sess);
        String childID = node.n.accept(this, arg);

        db.add(PredicateFile.NUM, ID, childID);
        db.add(PredicateFile.FILEPOSITION, ID,
                String.valueOf(node.lineno),
                String.valueOf(node.col_offset));

        return ID;
    }

    @Override
    public String visit(Literals.Int node, Integer arg) {
        String ID = _rep.getScopedID(node, scope.peek(), sess);

        db.add(PredicateFile.INT, ID, node.n_str);

        return ID;
    }

    @Override
    public String visit(Literals.Float node, Integer arg) {
        String ID = _rep.getScopedID(node, scope.peek(), sess);

        db.add(PredicateFile.FLOAT, ID, String.valueOf(node.n));

        return ID;
    }

    @Override
    public String visit(Literals.Complex node, Integer arg) {
        String ID = _rep.getScopedID(node, scope.peek(), sess);

        db.add(PredicateFile.COMPLEX, ID, String.valueOf(node.n),
                                          String.valueOf(node.i));

        return ID;
    }

    . . .

    @Override
    public String visit(ControlFlow.While node, Integer arg) {
        String ID = _rep.getScopedID(node, scope.peek(), sess);

        String testID = node.test.accept(this, arg);

        db.add(PredicateFile.WHILE, ID, testID);
        db.add(PredicateFile.FILEPOSITION, ID, String.valueOf(node.lineno),
                                               String.valueOf(node.col_offset));

        for (int i = 0; i < node.body.length; i++) {
            String nodeID = node.body[i].accept(this, arg);
            db.add(PredicateFile.WHILEBODY, ID, String.valueOf(i), nodeID);
        }

        for (int i = 0; i < node.orelse.length; i++) {
            String nodeID = node.orelse[i].accept(this, arg);
            db.add(PredicateFile.WHILEORELSE, ID, String.valueOf(i), nodeID);
        }

        return ID;
    }


    . . .

    @Override
    public String visit(Definitions.FunctionDef node, Integer arg) {
        /* Get ID before pushing to scope */
        String ID = _rep.getScopedID(node, scope.peek(), sess);

        _rep.addScopeRep(node, scope.peek(), node.name);

        /* Change scope to visit children */
        scope.push(node);

        String argsID = node.args.accept(this, arg);

        String returnsID = NULL_STR;
        if (node.returns != null)
            returnsID = node.returns.accept(this, arg);

        db.add(PredicateFile.FUNCTIONDEF, ID, node.name, argsID, returnsID);
        db.add(PredicateFile.FILEPOSITION, ID, String.valueOf(node.lineno),
                                                String.valueOf(node.col_offset));

        for (int i = 0; i < node.body.length; i++) {
            String nodeID = node.body[i].accept(this, arg);
            db.add(PredicateFile.FUNCTIONDEFBODY, ID, String.valueOf(i), nodeID);
        }

        for (int i = 0; i < node.decorator_list.length; i++) {
            String decoratorID = node.decorator_list[i].accept(this, arg);
            db.add(PredicateFile.FUNCTIONDEFDECORATORS, ID, String.valueOf(i),
                                                                     decoratorID);
        }


        /* Pop scope before returning */
        scope.pop();

        return ID;
    }

    @Override
    public String visit(Misc.Module node, Integer arg) {

        _rep.addScopeRep(node, node, filename);

        scope.push(node);
        super.visit(node, arg);
        scope.pop();

        return null;
    }
}
```

You probably just skipped over it so let's take some time to explain what is happening.

- Firstly, as expected, we extend the ***DepthFirstVisitor*** class and set our return and argument types to String and Integer accordingly. Arguments are of no importance, but the String return value is necessary for our implementation as we will soon see.
- The fields of our fact generation visitor are explained below:
  - ***filename*** is the name of our original Python source file. We use it in our node IDs described later.
  - ***db*** which is an instance of the ***Database*** class, provides an abstraction (along with an enumeration we define in another utility file) such that we can write to appropriate fact files in this fashion : ***db.add([EnumConstant], [Column1], [Column2], ...)***. Each enumeration constant corresponds to a separate ***Writer*** instantiated to write to a specific fact file. We can write arbitrarily many columns since ***add*** uses ***Varargs***.
  - ***_rep*** of type ***ASTRepresentation*** provides a mapping between ***Node*** objects and the ***scoped IDs*** we create to identify them. ***Scoped IDs*** are unique IDs that also include scope information. For example a node's scoped ID could be : ***input.py/foo/Int/219***. It basically consists of two parts : 1) The scope : ***input.py/foo***, 2) Our node's ID : ***Int/219***. Note that we make no distinction between function and class scopes but it is an easy addition to our implementation. In that sense, ***foo*** could either be a class or a function. The node's ID includes the node type (***Int*** here) and a unique number.
  - ***sess*** of type ***Session*** provides the unique numbers that constitute the last part of an ID. For simplicity, we just count the nodes as a means of providing unique IDs. The first node we form an ID for will get 0, the next 1, and so on. The IDing scheme can change by modifying the ***Session*** class implementation. More generally, in case we analyze multiple files (which we will not discuss in this thesis), we can arbitrarily reset the identification, or use the same ***Session*** object for all files.
  - ***scope*** of type ***Stack<Node>*** allows us to keep track of the current scope during our traversal, so that we can always provide the ***scoped IDs*** we discussed earlier.
  - ***NULL_STR*** is a ***String*** that defines what we output in our fact file when a field of our node is empty (***null***).

- We will now explain what typically happens when we visit a node :
  - First of, we ask for a scoped ID for our node.
  - Then, in case our node has subnodes, we call accept on them so that that they can get IDed. Their IDs are returned and saved by the original parent node.
  - The next step is writing to the fact files. Usually nodes only write to two fact files : 1) The fact file corresponding to their node category, 2) The fact file that holds positional information for all nodes that carry it. Some nodes only write to 1), but other nodes, specifically nodes that contain arrays of subnodes, write to additional fact files, one for each subnode array, so that the array contents can be preserved and correctly indexed. There are also nodes that don't write to fact files.
  - The final step is to simply return the node's ID we formed in the first step, to be used in case the current node is a subnode of another node.

  Important Note : Step 2 happens before step 3 because when we write to our current node's fact file we need to correlate it to its subnodes, so we must know their IDs.
- Another thing to further clarify is the form of the fact files. This pattern is followed when writing to fact files: When we write to the fact file that corresponds to our node's category, we always start with the node's ID and then write all fields that are single nodes (in this case their ID is written) or non-node fields (in that case we write their values as strings). If our node contains an array of subnodes we write to a new fact file for every array. The new fact files that correspond to the array in most if not all cases follow this pattern : (***originalNodeID, elementNumber, elementID***), where originalNodeID is the ID of the node containing the array, elementNumber is the position of the subnode in the array and elementID is the ID of the subnode.
- Notice how nodes that influence scope, like ***FunctionDef*** (line 116), push onto the scope stack to correctly keep track of the scope before they visit their subnodes and pop the scope stack before they return.

That was the outline of how our simple fact generation visitor works.

4.4 Summary

What we essentially did in this section, was generate facts about the AST's structure as the simplest fact generation case. The argument, is that given the framework we have constructed so far, we could implement arbitrarily complex fact generation visitors or even program transformers that actually modify the AST. As we mentioned time and again, the visitor pattern allows us to perform arbitrary operations on the tree so using that we could achieve any desired fact generation or transformation.

5. EXAMPLE WALKTHROUGH

5.1 Overview

In this section, an example that goes over the whole process described in previous sections is given. We have chosen to give our example on a trivial Python program, so that the complexity of the AST does not clutter our description of what happens in every step. Moreover, choosing a trivial program helps us elaborate on details, whereas with a larger, more complex program, a full description and analysis would be stupidly lengthy if not impossible. It has already been demonstrated that a mere Python function definition yields a substantially lengthier JSON representation of the AST [2].

5.2 Example

5.2.1 Initial source code

In the next listing, we give our initial source code, assuming it is stored in the file ***example.py***. We will simply perform a trivial addition of two integer numbers.

Listing 22: Example Source Code

```
1 + 2
```

5.2.2 AST in JSON format

Following the first steps described in section [3], the AST is exported to the file ***example.py.json***.

Listing 23: astexport command for example.py

```
astexport -p < example.py > example.py.json
```

The contents of the file ***example.py.json*** are now given.

Listing 24: Example AST in JSON

```
{
    "ast_type": "Module",
    "body": [
        {
            "ast_type": "Expr",
            "col_offset": 0,
            "lineno": 1,
            "value": {
                "ast_type": "BinOp",
                "col_offset": 0,
                "left": {
                    "ast_type": "Num",
                    "col_offset": 0,
                    "lineno": 1,
                    "n": {
                        "ast_type": "int",
                        "n": 1,
                        "n_str": "1"
                    }
                },
                "lineno": 1,
                "op": {
                    "ast_type": "Add"
                },
                "right": {
                    "ast_type": "Num",
                    "col_offset": 4,
                    "lineno": 1,
                    "n": {
                        "ast_type": "int",
                        "n": 2,
                        "n_str": "2"
                    }
                }
            }
        }
    ]
}
```

5.2.3 Simplified JSON & Relevant Java Classes

At this point, we will not oversimplify as we did in [3]. Instead, we only disregard the outermost part of the JSON, the object with type ***Module*** that contains in its body all ”first level” nodes in our program. We will not be generating facts for the ***Module*** node since we focus on analyzing one file at a time, and moreover, our example only contains one expression. The rest of the objects' fields are only reordered to promote readability.

Listing 25: Example AST in JSON (Simplified)

```
{
    "ast_type": "Expr",
    "lineno": 1,
    "col_offset": 0,
    "value": {
        "ast_type": "BinOp",
        "lineno": 1,
        "col_offset": 0,
        "left": {
            "ast_type": "Num",
            "lineno": 1,
            "col_offset": 0,
            "n": {
                "ast_type": "int",
                "n": 1,
                "n_str": "1"
            }
        },
        "op": {
            "ast_type": "Add"
        },
        "right": {
            "ast_type": "Num",
            "lineno": 1,
            "col_offset": 4,
            "n": {
                "ast_type": "int",
                "n": 2,
                "n_str": "2"
            }
        }
    }
}
```

The relevant class definitions are given below to reveal the structural correlation of the JSON and the object graph we construct when importing and deserializing the JSON. The definitions of some of the classes have already been provided, but are repeated for coherence. The accept method described earlier, that allows the visitor pattern to operate, is excluded for economy of space.

Listing 26: Expr Class Definition

```java
public class Expr extends Node {
    public int lineno;
    public int col_offset;

    public Node value;
}
```

Listing 27: BinOp Class Definition

```java
public class BinOp extends Node {
    public int lineno;
    public int col_offset;

    public Node op;
    public Node left;
    public Node right;
}
```

Listing 28: Num Class Definition

```java
public class Num extends Node {
    public int lineno;
    public int col_offset;

    public Node n;
}
```

Listing 29: Int Class Definition

```java
public class Int extends Node {
    public long n;
    public String n_str;
}
```

Listing 30: Add Class Definition

```java
public class Add extends Node {}
```

The 1-to-1 correspondence between the JSON and the class definitions should be apparent.

5.2.4 Java Object Graph

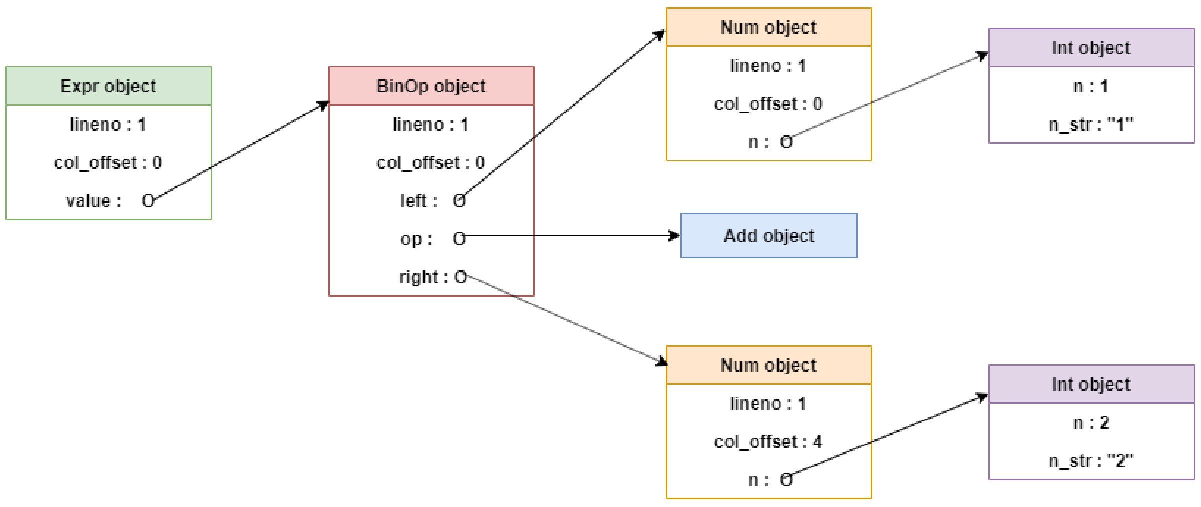

Figure 1: Object Graph for our example

The object graph in figure [1] helps visualize the object structure residing in RAM after having imported and deserialized the file ***example.py.json***. Mentally tilting the graph by 90 degrees to the right better reveals the object graph's tree shape. In essence, the AST now resides in our Java program's memory with the above structure.

5.2.5 Visiting the Object Graph

The relative visit functions of our fact generation visitor that are triggered when visiting the object graph in figure [1] are now presented. Again, some of them have been presented earlier, but are repeated for coherence.

Listing 31: Expr Visit Method

```java
@Override
public String visit(Expressions.Expr node, Integer arg) {
    String ID = _rep.getScopedID(node, scope.peek(), sess);

    String valueID = node.value.accept(this, arg);

    db.add(PredicateFile.EXPR, ID, valueID);
    db.add(PredicateFile.FILEPOSITION, ID,
                                String.valueOf(node.lineno),
                                String.valueOf(node.col_offset));

    return ID;
}
```

Listing 32: BinOp Visit Method

```java
@Override
public String visit(Expressions.BinOp node, Integer arg) {
    String ID = _rep.getScopedID(node, scope.peek(), sess);

    String leftID = node.left.accept(this, arg);
    String op = node.op.accept(this, arg);
    String rightID = node.right.accept(this, arg);

    db.add(PredicateFile.BINOP, ID, leftID, op, rightID);
    db.add(PredicateFile.FILEPOSITION, ID, String.valueOf(node.lineno),
                                           String.valueOf(node.col_offset));

    return ID;
}
```

Listing 33: Int Visit Method

```java
@Override
public String visit(Literals.Int node, Integer arg) {
    String ID = _rep.getScopedID(node, scope.peek(), sess);

    db.add(PredicateFile.INT, ID, node.n_str);

    return ID;
}
```

Listing 34: Num Visit Method

```java
@Override
public String visit(Literals.Num node, Integer arg) {
    String ID = _rep.getScopedID(node, scope.peek(), sess);
    String childID = node.n.accept(this, arg);

    db.add(PredicateFile.NUM, ID, childID);
    db.add(PredicateFile.FILEPOSITION, ID,
            String.valueOf(node.lineno),
            String.valueOf(node.col_offset));

    return ID;
}
```

Listing 35: Add Visit Method

```java
@Override
public String visit(Expressions.Add node, Integer arg) { return "+"; }
```

By examining the above methods - and more specifically the ***db.add()*** calls - one can infer the structure of the fact files produced for these node types. Further explanation about the operation of these methods was given in section [4.3].

A summary of the structure of the above mentioned fact files follows.

Table 1: Fact Files' Structure

File Name	Columns
FilePosition.facts	(nodeID, lineno, col_offset)
Expr.facts	(nodeID, valueID)
BinOp.facts	(nodeID, leftID, op, rightID)
Num.facts	(nodeID, nID)
Int.facts	(nodeID, n_str)

Something that should be pointed out - and was only briefly touched upon earlier in chapter [4] - is that some nodes' visit methods do ***not*** write to fact files (e.g. [The Add visit method], just presented). This is a choice that was made to limit the volume of fact files we produce since in the case of operators (e.g. $+, -, *$) it is probably wasteful to view them as nodes and ID them, and thus provide separate fact files for them. In our implementation, we simply return the corresponding operator as a string which is subsequently written directly to the fact file of the node that contains the operator node as a field.

5.2.6 Fact Files

Having exported our AST into JSON, and given that as input to our Java program, which deserialized it and called our fact generation visitor on it, we get the aforementioned fact files as an output. In this section the contents of these files will be listed.

Listing 36: FilePosition.facts

```
NODE_ID                 LINENO  COL_OFFSET
example.py/Num/2        1       0
example.py/Num/4        1       4
example.py/BinOp/1      1       0
example.py/Expr/0       1       0
```

Listing 37: Expr.facts

```
NODE_ID                 VALUE_ID
example.py/Expr/0       example.py/BinOp/1
```

Listing 38: BinOp.facts

```
NODE_ID                 LEFT_ID                 OP      RIGHT_ID
example.py/BinOp/1      example.py/Num/2        +       example.py/Num/4
```

Listing 39: Num.facts

```
NODE_ID                 N_ID
example.py/Num/2        example.py/Int/3
example.py/Num/4        example.py/Int/5
```

Listing 40: Int.facts

```
NODE_ID             N_STR
example.py/Int/3    1
example.py/Int/5    2
```

The IDing scheme was described in section [4]. The correspondence between the [object graph] and these fact files is pretty simple. Every object instance corresponds to a row in the appropriate fact file (with the exception of the ***Add*** node as discussed earlier). The member fields of the object (minus the positional information that is written to a separate file) are directly written to the fact file if they are not references to other objects but primitives, while for fields that are references, the ID of the referenced object is written so as to indirectly point to that object.

5.3 Summary

The example given in this section sheds light on the whole process. The starting, intermediary and final files/structures were thoroughly presented. To summarize once more :

$$Python\ program \xrightarrow{astexport} JSON\ AST \xrightarrow{Gson} Java\ Objects \xrightarrow{Visitor} Fact\ Files$$

6. ADDITIONAL THOUGHTS

6.1 Verification & Loss of Information

An important question that arises is the following : Are the products of our transformation process at all steps equivalent to the original Python AST? The short answer is : probably yes. The lengthier answer is that we assume the JSON representation is equivalent, since it is produced by an external tool, namely ***astexport***. Additionally, our Java object representation of the tree is equivalent to the JSON representation. That was tested by re-serializing our Java objects to JSON and comparing to the original JSON we used as input. Finally, the way we constructed the fact files, we have probably preserved all the necessary information contained in our objects. Consequently, in theory, we should be able to reconstruct the original Python AST or even further, the original Python program. This is not an easy task however, especially when starting from the last product of the chain, fact files. Proving that starting from fact files we can reacquire the original program would be interesting but it is not explored in this thesis. It could possibly be examined as future work.

We state, with little uncertainty, that all the information of the original AST or program (they are equivalent since Python uses the AST to produce the bytecode that is interpreted) are preserved throughout our transformation. This is merely a statement however. A proof would require rebuilding the AST or program starting from the fact files as discussed in the above paragraph.

6.2 Implementation Language

In retrospect, we need not have implemented our work using the Java language. A Python implementation might have been a lot less troublesome since most of the transformation steps would be eliminated, and as a result we would have only had to implement the visitor pattern to perform fact generation. There are two main reasons for the language choice : 1) Out-of-the-box performance of the JVM compared to the Python interpreter, 2) Integration with the Doop [1] platform.

6.3 Caveats

As stated in Python's documentation, its abstract syntax grammar might change in an arbitrary way with every release, thus affecting the structure of the AST (via addition, removal or modification of nodes of different types), and subsequently rendering the work described here incomplete or imprecise. Usually, changes are of a small scale, so, a few modifications of our implementation could allow us to work with any desired Python version. For the purposes of this thesis, we have only considered the AST structure as it is described for Python 3.6.

7. CONCLUSIONS

The number of static analysis tools and techniques is rapidly growing, not only for the Python language, but universally. Checking the source code manually is an almost impossible task in large codebases, with the software becoming more and more complex and lengthy. In this thesis, we provided a framework upon which various static analyses and program transformations could be implemented for Python programs, on the AST level. Our Java program, along with our tools, [astexport, Gson], transforms the input Python program to equivalent fact files. The fact files can then be imported for analysis. For example, a Datalog program could be used to write a declarative static analysis algorithm, as is done in the Doop framework. Moreover, our Java program is extensible, thus new algorithms on the AST are easy to implement through the visitor pattern we provide.

This work is by no means complete. It is merely the first step towards a Java analysis framework for Python programs, and subsequently can be extended in various ways:

- Reconstruct the original program from fact files to prove that the transformation is non-destructive and reversible.
- Implement different kinds of visitors to suit various traversal/modification needs.
- Write more complex fact generation visitors that produce non-trivial facts.
- Construct static analysis programs in Datalog that use the facts produced as input.
- Write Datalog programs to check the consistency of the fact files. (e.g. Check if all IDs are valid)
- Support multiple Python versions. (we only focused on 3.6 while there is still a lot of code in Python 2.7)
- Add parallelism wherever possible and the ability to produce facts for multiple files concurrently.

ACRONYMS AND ABBREVIATIONS

AST	Abstract Syntax Tree
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
EDB	Extensional Database
IDB	Intensional Database

REFERENCES

[1] Doop Project Repository [Online]
`https://bitbucket.org/yanniss/doop`

[2] Doop Tutorial, PLDI 2015 [Online]
`https://plast-lab.github.io/doop-pldi15-tutorial/`

[3] Python AST Documentation [Online]
`https://greentreesnakes.readthedocs.io/en/latest/nodes.html`

[4] Python Statistics [Online]
`https://www.jetbrains.com/lp/devecosystem-2019/python/`

[5] Python ast Module [Online]
`https://docs.python.org/3.6/library/ast.html`

[6] astexport library [Online]
`https://github.com/fpoli/python-astexport`

[7] Gson library [Online]
`https://github.com/google/gson`

[8] Visitor Pattern [Online]
`https://en.wikipedia.org/wiki/Visitor_pattern`