**Abstract :**

**Introduction :**

The technique of finding and removing code segments from a software programme that are not being used or accessible during programme execution is known as dead code detection. Memory and resources are used by unused code, which impacts how effectively a programme runs. Software can be made more efficient by freeing up resources and streamlining execution by eliminating dead code. Extraneous code adds complexity to the codebase, which makes it more difficult to comprehend and manage. Additionally, unused code can lead to security holes, particularly if it is out-of-date or has security issues. Therefore, dead code discovery is essential for raising the caliber of software, optimizing performance, and enabling a codebase that is

more effective and manageable.

Finding unneeded code by hand takes a lot of effort, is laborious, prone to mistakes, and necessitates in-depth understanding of the source. the confident and automated elimination of oxbow code from Erlang codebases, therefore actively improving the codebases' quality and maintainability **[1].** Eliminating clones might result in a reduction in software maintenance expenses of up to 50%. Clones might be eliminated mechanically using in-line processes or conventional preprocessor macros **[2].** A tool for swiftly comparing the source code of several C programme versions was given by the authors. The method can track minor modifications to global variables, types, and functions and is based on partial abstract syntax tree matching. displayed results based on widely used open source software, such as the Linux kernel, BIND, OpenSSH, Apache, and Vsftpd **[3].** The domain was formalized, (abstract) syntax was established, and semantics were defined using semantic equations of denotational semantics by the author's inquiry. also examined how a domain-specific language may be used to develop a query language for Python ASTs, and how one evaluation method for such a DSL would be to compile it using the "reduce" method **[4].** Provided a Java back-end for the programme ApiGen, which creates abstract syntax tree implementations. This work's contribution is the combination of creating a Java data structure that is tightly typed and has maximum sub-term sharing **[5].**

The term "code smell" describes specific patterns or traits in code that point to possible problems, flaws, or places where software development has to be strengthened. Defines the argument list, duplicate code, dead function, etc. as well. Code smells, security flaws, possible mistakes, and compliance with best practices or coding standards are all checked for in the code using static analysis tools. Finding code smell and their static analysis is our primary objective.

In section 1, illustrated brief discussion about the dead code detection based on the memory and resources. Section 2 shows the related works on dead code detection. In section 3, we discussed the goal and the motivation of this paper.

**Literature review :**

**3. Methodology:**

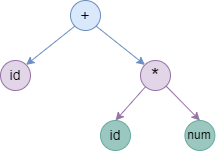
**3.1. Deciphering Python: Using (AST) to understand the code**

The Python interpreter follows a predictable set of steps to translate code into instructions that a machine can run. The code is parsed into a list of pieces usually called tokens. These tokens are based on a set of rules. The list of these tokens is transformed to build an abstract tree.

**3.2. Abstract Syntax Tree**

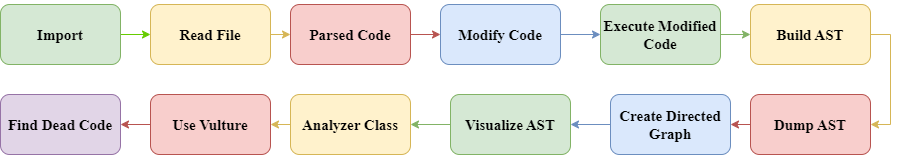
An Abstract Syntax Tree is a collection of nodes which are linked together based on the grammar of the Python language. From an abstract syntax tree, the interpreter can produce a lower level form of instructions called bytecode. With the bytecode instructions available, the interpreter can finally run the code. The bytecode is used to call functions in the operating system. Bytecode is very primitive and very tuned to making the interpreter fast. In other words, bytecode is designed for computers over people. Abstract syntax trees have enough structured information within them to make them useful for learning about the code.

**Structure of AST:** Each node in the AST represents a specific element of the program’s syntax, such as functions, statements, and expressions. AST nodes have a defined structure, consisting of a type, attributes, and child nodes.



**Figure 1: Structure of AST Tree**

**3.3. Process Flow**



**Figure 2: Flow of working process of code**

**3.4. Implementation**

**3.4.1. Import Modules**

The import modules are,

* **“import ast”** for analyzing the file of code by ast and
* **“import vulture”** for identifying the dead code of the file.
* **“from graphviz import Digraph”** for visualizing the AST tree as a directed graph.

**3.4.2. Read the file**

Read a Python file’s text (in this case, testcode.py). In this python file there exist some functions like addition which can add two numbers, subtraction which can subtract two numbers, difference which can subtract two numbers with absolute value, multiplication which can multiply two numbers. Also there are some import files that are never used and some unused variables. The calling functions are “multiplication” and “difference”.

**3.4.3. Parse the code for making AST**

To generate the AST tree we have used the ast\_parse() function. And for exploring the tree we have used the ast.walk() function. This will create an AST tree that represents the structure of the code, with each node in the tree corresponding to a specific Python language construct, such as a

function definition or a call expression. We have also used the ast.dump() function to visualize the AST tree in a more readable format.

Each element of the code is represented by a node in the AST.

* The outermost node is the “module” node, which represents the entire module.
* The “function def” node has various attributes, such as the name of the function and the arguments it takes.
* The body of the function is represented by the list of nodes which is denoted as “Expr” node.
* “BinOp” (left, op, right) is defined in the ast module that is used to express binary operations.
* “alias” (name) is used to express an alias second name for a piece of data

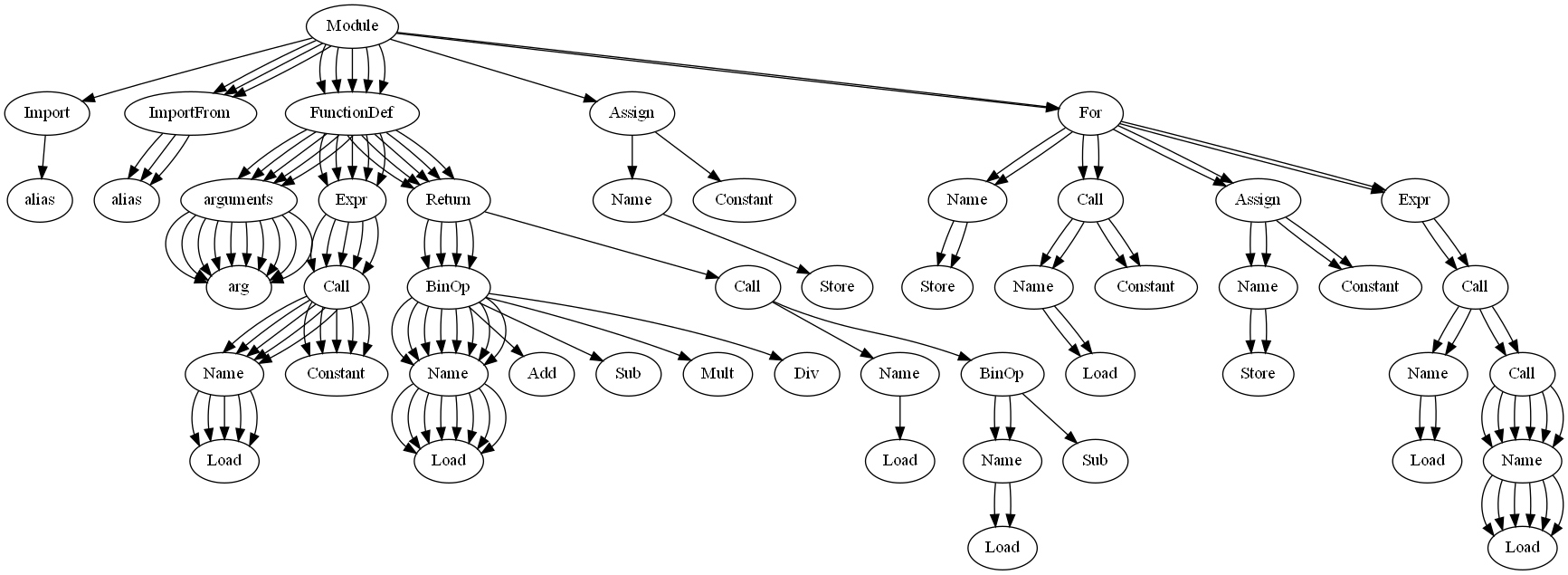
**3.4.4. Visualization of AST Tree**

We employ a graphviz Digraph object to store the nodes and edges of the Abstract Syntax Tree (AST). To populate the Digraph, we define a recursive function named build\_ast\_graph, responsible for adding child nodes, their parent nodes, and edges.

Subsequently, we invoke build\_ast\_graph on the AST's module node. Following this, we specify the format, utilize the render method, and generate a PNG file to visually represent the Digraph. Upon opening the PNG file, one can observe a graphical depiction of the AST, where nodes symbolize various code elements (e.g., functions, statements, expressions), and edges illustrate their interrelationships.

Executing the code and rendering the Digraph as a PNG file produces a visual representation, facilitating an understanding of the code's structure, encompassing functions, statements, expressions, and their respective relationships and hierarchies.

* **The AST Tree of specific code:**



**Figure 3: AST Tree of specific code:**

**3.4.5. Use analyzer Class for unused import file**

The analyzer follows the visitor pattern to extract information out of the tree.

* Class Analyzer(ast.NodeVisitor) is a class with two functions.
* def visit\_Import() find the import
* def visit\_ImportFrom() find the import from
* {'from': ['Digraph', 'Image', 'pprint'], 'import': ['ast', 'vulture']} These are the import and import from in the python test file.

**3.4.6. Vulture**

Vulture is a python library that is used to identify the dead code. Dead code is a piece of code that can not be executed during runtime.

* $ pip install vulture

**Types of unused code in vulture**

| **Code Type** | **Confidence Value** |
| --- | --- |
| class argument, unreachable code | 100% |
| import | 90% |
| attribute, class, function, method, property, variable | 60% |

**Working way of vulture**

Vulture uses the ast module to build abstract syntax trees for all given files. While traversing all syntax trees it records the names of defined and used objects. Afterwards, it reports the objects which have been defined, but not used. This analysis ignores scopes and only takes object names into account. Vulture sorts unused code by its number of lines.

**5. Result and Analysis**

To detect the file of code we have used the vulture and we get the unused functions, unused variables, and unused import with confidence value and sorts by number of lines of code.

* **!vulture testcode.py**

| **Line number of code** | **Code Type** | **Name** | **Confidence Value** |
| --- | --- | --- | --- |
| 1 | unused import | ast | 90% |
| 2 | unused import | dgph | 90% |
| 3 | unused import | Image | 90% |
| 4 | unused import | pprint | 90% |
| 6 | unused function | addition | 60% |
| 9 | unused function | subtraction | 60% |
| 18 | unused function | division | 60% |
| 24 | unused variable | variableDigit | 60% |
| 27 | unused variable | variableNewDigit | 60% |

After applying vulture to our file and analyzing the code we have found the line number of code in the file which is sorted properly. The unused function, unused import and unused variable has also been printed with their name. Finally, we got the confidence value of unused code which is the same as vulture analyzing confidence value.

**Conclusion :**

**Reference :**

[1] Rodríguez, F.B. and Castro, L.M., 2021, August. Detecting oxbow code in Erlang codebases with the highest degree of certainty. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang* (pp. 28-40).

[2] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L., 1998, November. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 368-377). IEEE.

[3] Neamtiu, I., Foster, J.S. and Hicks, M., 2005, May. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories* (pp. 1-5).

[4] Arts, E.M.A., 2022. Towards Querying Abstract Syntax Trees for Python Programs.

[5] Van den Brand, M., Moreau, P.E. and Vinju, J., 2005. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings-Software*, *152*(2), pp.70-78.