JTE HCMUTE
ISSN: 2615-9740

JOURNAL OF TECHNICAL EDUCATION SCIENCE
Ho Chi Minh City University of Technology and Education
Website: https://jte.edu.vn
Email: jte@hcmute.edu.vn

Development of Code Evaluation System based on Abstract Syntax Tree

Anh-Tu Phuong Nguyen, Van-Dung Hoang*
Ho Chi Minh City University of Technology and Education, Vietnam

*Corresponding author. *Email: dunghv@hcmute.edu.vn*

ARTICLE INFO

Received: 02/01/2024
Revised: 22/02/2024
Accepted: 22/02/2024
Published: 28/02/2024

KEYWORDS

Structure Code;
AST;
Evaluating code in learning online;
Global state and Compile;
Hashing value.

ABSTRACT

Analyzing and evaluating student-generated code poses a big headache for programming education. Code evaluation is a delicate task requiring accuracy, efficiency, and error-checking. This paper considers one of the many useful tools in code evaluation. It's Abstract Syntax Trees (AST). AST is a robust tool for reading, filtering, and weighing student code. It is built from tree regular expressions for common programming patterns. This paper considers, firstly, integrating AST into code evaluation projects. Secondly and most importantly to the present implementation work is that Python programming language provides excellent compatibility for implementing such a program. Taking advantage of Python's feature that allows the inputting of code as a string, AST records and tracks everything about the user's own source. With AST being integrated into platforms for education, how to evaluate student code has been completely altered; as sites such as LeetCode and HackerRank testify. This paper presents an accurate, efficient and error-aware approach to code evaluation by improving on educational websites incorporating AST. Adapting well to the dynamics of programming education, this elaborate assessment system will provide students with truthful assessments and profound feedback on their coding capabilities. This article shows how AST-based evaluation has revolutionized the way programming is taught, and thoroughly explores its role in evaluating code.

Doi: https://doi.org/10.54644/jte.2024.1514

1. Introduction

This introduction of students to the dynamic world of programming education offers numerous challenges, one of which is code assessment. Code efficiency is vital not just for the learning process, but also for evaluating possible future programmers. Because coding is abstract and multithreaded, it needs an expert system capable of comprehending the maze immobilizing student-generated scripts. Under these conditions, the usage of Abstract Syntax Trees (AST) becomes essential to improve both the precision and efficiency with which code may be evaluated.

Because AST employs some tree of regular expressions to encode typical patterns of programming language, it is essential for understanding, extracting, and assessing student code. It is an important part of code evaluation, efficiency analysis and compilation error detection. As students attend lectures and do coding exercises, the correctness of code assignment becomes increasingly important. Beyond correctness, AST allows educators to accurately, and deeply examine student code on elements such as efficiency or optimization. Integrating AST is most advantageous when student code is submitted to a mix of test cases, giving students an overall score for their ability to handle different kinds of programming tasks. Because Python is an interpretive language with built-in support for AST, this paper uses it as the main programming language. Having Python process a string input user code allows for an adaptable incorporated approach in which everything about the users 'code is recorded and monitored. Because of its integrability with educational platforms and popularity in modern programming languages, AST is changing the way student code gets graded. For example, online platforms such as LeetCode and HackerRank have successfully implemented AST-based code evaluation which gives students very exact and informative feedback.

2. Related Work

It's a daunting task to teach students the thought process required for successful software engineers. Though code can be implemented using a linear approach, the mental construction and process of problem solving typically is not linear and demands to have a clear understanding of class structures, control flow and so on. Prior to any code can be written. This concept can be difficult to beginning CS students to grasp and apply to their own coding projects. We provide a visualization that helps students to more be able to comprehend the coding process based on the existing study [1]. The keystroke data collected is incorporated into an Abstract Syntax Tree (AST), creating an AST that is Temporal. We give the required information to prove that this visualization exists in every project that requires keystroke data. Additionally, we provide a reference to another type of keystroke visualization known as the Code Process Chart (CPC) which was the source of inspiration for this temporal-based AST. It is our goal to use the temporal ASTs along with their CPCs to improve the understanding of students about coding [2]. There is a study about augmented reality software application called AST[AR] aimed at educating novice programmers on data structures using AST. Through a user experience study involving 30 volunteers, the application garnered positive feedback from 80.6% of the participants [3]. Software repositories can be explored in terms of source code to help to understand the process of developing software. This tool can compare source code from different C application versions. The method is based on partial AST to keep track of modifications that are made to global variables as well as functions. The method can be applied to more complicated questions, by examining the different aspects of software growth. We look at how these can guide the design and development of a dynamic updating system for software. Tests performed with different versions of Open-Source Software form the foundation for our findings. BIND, OpenSSH Apache, Vsftpd and Vsftpd, and Vsftpd and the Linux Kernel [4]. Control-flow is also a big problem that needs to be considered. Following the study uses CFAST to analyze how students approach programming tasks globally. It identifies different control-flow designs, how quickly students recognize correct structures, and the effort to convert these to functional programs. Despite many CFASTs from simple problems, most student work fits into a few structures [5]. On the other hand, we also focus on improving code summarization based on block-wise Abstract Syntax Tree Splitting (BASTS). Extensive experiments on benchmarks demonstrate that BASTS outperforms existing methods across various evaluation metrics [6].

This testing method is useful to design and evaluate the existing test suites. Though mutation testing, which has been proven to be effective over the many years with various types of software platforms, it hasn't been widely used in the industries. The main reason that hinders testers and developers from utilizing mutation tests is high computation cost. It is the necessity of manually recognizing similar mutations, which causes significant delay and also increases the intensity of labor. We design and develop an abstract syntax tree Recurrent Network model which can automatically identify the same mutants during mutation testing. A pilot study conducted by the authors showed that the method based on machine learning proved to classify identical mutants with greater accuracy than 90% [7]. This method can greatly reduce the time and effort used to identify similar mutants in the course of mutation testing. Stack Overflow Code Snippets provides a wealth of information regarding small source code. Text is enhanced by a natural conversational language. These code fragments that are annotated could also serve as the basis for automated software to answer natural language queries [8]. There is a new software reliability model which is based on Syntax tree, aimed at analyzing system components' importance measures, utilizing source code to generate a syntax tree, and subsequently constructing a reliability model. The model's creation process and its ability to calculate system characteristics, like important measures, are outlined [9]. Moreover, we also got the introduction about PSIMINER, a tool designed to process PSI trees derived from the IntelliJ Platform. PSI trees encapsulate code syntax trees and associated functionalities, enabling the enrichment of code representation through static analysis algorithms available in modern IDEs. The tool demonstrates its utility by inferring identifier types in Java ASTs and extending the code2seq model for predicting method names, showcasing the potential of using PSI trees for enhancing machine learning-based code analysis [10].

Web applications have quickly become an indispensable way of providing all services through the Internet and their numbers have rapidly expanded. Unfortunately, when created without appropriate testing protocols or experience they pose significant risk. Web application vulnerabilities arise through

design flaws within online applications that allow attackers to gain entry and take control over any internal objects not intended for them, potentially compromise its integrity, alter data or steal confidential information from it. This system's goal is to detect web application vulnerabilities before they are exploited by an attacker by employing Python 3.7 built-in tools like AST, CFG, Flask and Django [11]. For this purpose, a special scanner was developed that utilizes these detection abilities. Nested Bigrams provides authorship attribution of unknown source code in cybersecurity with over 90% classification accuracy [12]. Testing deep learning libraries is crucial to guarantee their quality and security when used for various applications involving deep learning technology. Enhancing AST decoding to real vectors using Tree-based Embedding Framework is another essential aspect of improving software defect prediction. One popular technique when employing an AST technique is TBE Framework Decoding of AST. This decoding method seeks to increase semantic understanding among ASTs, therefore making cross-project defect prediction (CPDP) tasks more accurate [13]. As differential testing is frequently employed to assist in developing test oracles, its ongoing maintenance poses new difficulties; to address this, we developed DiffWatch as a fully automated Python tool which finds differential test practices within DLLs as well as keeping an eye on changes made to external libraries that might cause these tests to need updating [14].

3. Research Approaches

AST lies at the core of Python programming. Whenever read or write a statement in code, an AST is being generated subtly and constantly behind back by some process; it's all taken for granted-nobody thinks anything more about this potency inside their program When properly represented as source code text on screen, every line refers to something that exists to developers, structural subtleties are: the nail's head The AST module of Python provides one keen eye to take a look. This essay takes an in-depth journey examining all layers of the AST: its vital constituents; a multitude of applications appointing it to take the role as shaper and decorator among all this partner generator's future design dream. Gaining a thorough understanding of AST is an important first step in our exploration. An AST is a hierarchical data structure which has the same structure as Python code syntax [15]. In order to generate an AST, source codes are first parsed by AST module of standard library or built in functions divide() and makettree(). These individual nodes represent some syntactic elements (code bases). If stolen then two trees can be added together. One aspect of hello content is captured by Python AST very well, with nodes (representing words or blocks) for assignment statements (assign), functions definitions: Funcdef and such control flow statements as Ifs, While-s etc. which provide insight into their logical connections in relationship to one another; while visualizations give an idea about how logic moves forward both visibly and. Python's AST: A meticulous documentation of program flow is at the heart of every Python program and this core relies upon an Abstract Syntax Tree. Some nodes, like BinOp and Allow are binary operations. Others can represent more than one operation with a single node such as Binop which is binary operations; some serve to call functions/methods -- Call and Name respectively- while others provide variable/function names--Variable and possible other branches of the articulated -arm model (cf.) Legal.~ Permits direct access to bindings from data flows AST offers an expansive representation that goes far beyond fundamental constructs in Python programs; its scope spans multiple elements: Import statements and ImportFrom Statements provide external intervention; ClassDef Namespaces are used to show how this stuff operates within the actual program, constructing units of greater depth. Constructs such as Attribute Look- up on an instance object appear inside a method definition itself (they have arrows pointing into function bodies).

ASTs are an abstraction of the source code and capture structural elements of code snippets. They represent these features in a tree-like structure. ASTs, as opposed to Concrete Syntax Trees, a type of syntax trees that gives an exact representation for source code is more appropriate to code snippet search because it represents source code on an abstract level [16]. Grammar symbols and other unimportant elements can therefore be dismissed as noise. For comparison, we only consider the general structure of source files and their meaning. Test development and test suite quality evaluation are aided by mutation testing [17]. It may be aimed toward programs written in many languages. The application of mutation testing is limited by its high cost. The efficient handling of them in the Python context was demonstrated. We go overusing generators to introduce first- and higher-order mutations into an abstract syntax tree, how to handle code coverage using AST, and how to execute mutants by injecting them into tests, picture

JTE HCMUTE
ISSN: 2615-9740
JOURNAL OF TECHNICAL EDUCATION SCIENCE
Ho Chi Minh City University of Technology and Education
Website: https://jte.edu.vn
Email: jte@hcmute.edu.vn

to present for this idea. Create a tool in this language that can identify test smells and analyze test smells empirically in Python projects. First, we compiled a list of test smells from previous studies, choosing those that were either similarly functional in Python's standard Unittest framework or could be regarded as language-neutral [18], then presenting this idea by picture. Accordingly, figure 1 shows a diagrammatic flowchart of solution what all we do.

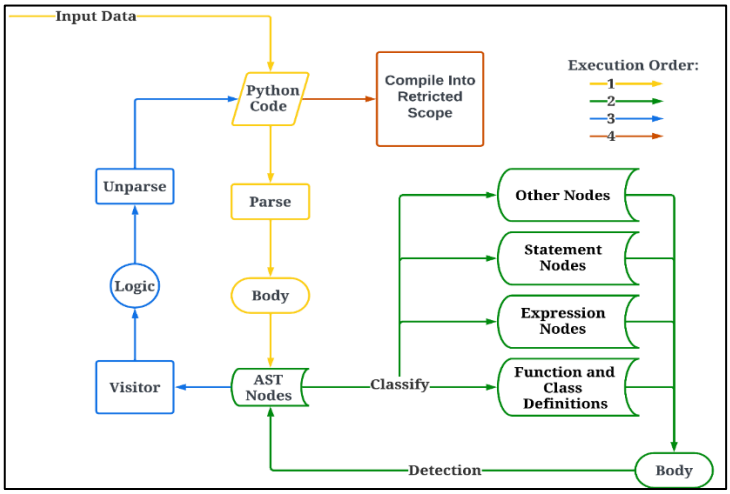

Figure 1. *Flowchart for AST Implementation Solution.*

NodeVisitor in Python includes a comprehensive AST module for reading and modifying Python ASTs [19]. An AST of this type is represented by a collection of objects that are linked together by their attributes. An if-statement, for example, is represented in Python as an instance of an AST. If class with test, body, and or else attributes for test condition, body-branch, and else-branch.21 These attributes can be changed directly to alter the AST. Python can then execute the program represented by that AST using this (possibly modified) AST. The Python standard library provides several methods for iterating over a Python AST. First off, the generator *AST.walk(x)* returns every descendant node of node x; in theory, this is done in any order. Second, by subclassing the NodeVisitor class, which produces nodes in depth-first order, the visitor pattern can be applied. Finally, by recursively iterating over a node's attributes (which represent child nodes), one can construct their own iterator. We will look at how these visitors can be used to look up our AST properties in this section. NodeVisitors and ASTs vary slightly depending on the Python version. The rest of this section is predicated on the usage of Python 3.10. The flowchart is illustrated in Figure 2.

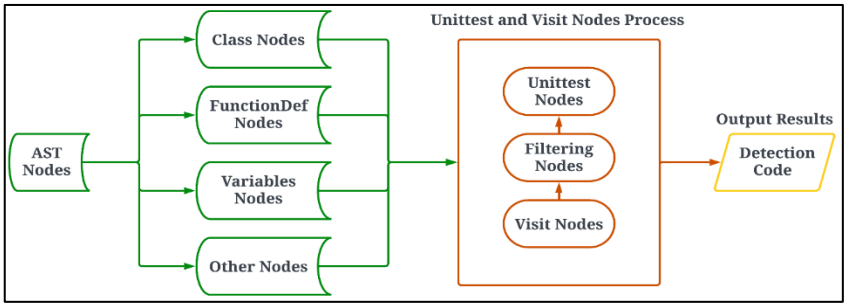

Figure 2. *Detail for Unittest and Visit Nodes Process.*

4. Implementation and Experimental Results

Understanding globals() method: This function in Python provides access to the global symbol table, an extensive dictionary listing all global variables and their values that define current global scope. With dparseynamically modifiable variables as its focus, the globals() function acts as a portal for dynamically retrieving these variables dynamically modifying or retrieving them - this paper will discuss its implementation, applications and possible benefits before delving deeper.

JTE
HCMUTE
ISSN: 2615-9740

JOURNAL OF TECHNICAL EDUCATION SCIENCE
Ho Chi Minh City University of Technology and Education
Website: https://jte.edu.vn
Email: jte@hcmute.edu.vn

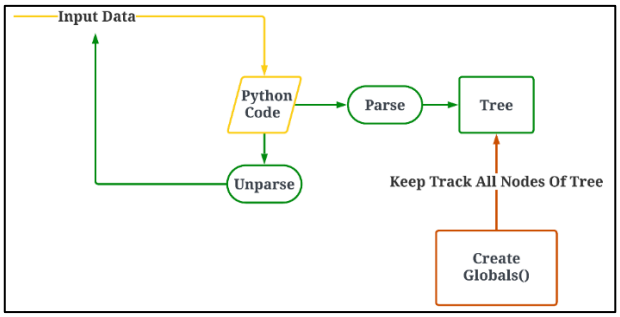

Figure 3. *Constructor for Process ClassBaseAST.*

Implementation ClassBaseAST: The constructor calls parse_code(), which initializes an AST representation of supplied Python code and thus sets up further AST operations [20]. ClassBaseAST's primary strength lies in translating Python into an AST and vice versa - this translation feature serves as its centerpiece. Parse_code() can create an AST representation of Python code strings using two parameters to specify their parsing mode (exec, eval, func_type or single). An optional Python code parameter and mode parameter must also be supplied to this function. If no code is supplied to this method, its default _python_code will be utilized by this technique. This technique parses and creates an AST using the AST module; several parsing modes can be selected through mode parameter allowing for different use cases. For later use, the generated AST is stored in the _python_code attribute for easy reference. An AST node can be converted back into Python code with unparse_code() for standalone and instance level conversions respectively. This technique uses AST.unparse() to convert an AST node into Python code string representation of its original structure; otherwise, it defaults to using stored Python _python_code for conversion. If an AST node is passed in as a parameter for transformation, this technique will convert that node directly; otherwise, stored _python_code will be converted by default [21].

Checking AST: This method uses AST.unparse() to convert an AST node into a Python code string representing its original structure, providing an excellent solution when performing operations without an applicable AST (i.e. isinstance(self._python_code, AST.AST). This can help prevent errors associated with such operations (isinstance(self._python_code, AST.AST)).

Additional Actions: ClassBaseAST contains two abstract methods, filter_node() and get_func(), for implementing specific AST manipulation tasks by its derived classes. Filter_node() lets grab just the AST nodes based on their type—usually it's used to pull out specific function definitions. It returns a list of the matching nodes. Then Get_func() is an easy way to lookup a function by name in an AST if it is needed to run it in a restricted namespace. The abstract methods there are handy because they let developers customize and expand the functionality. However, they want if the base methods don't quite fit their needs. Basically, it gives flexible building blocks to find and subset AST nodes.

EvalCode: In this task, we implement some additional Actions:

Filter_node() of EvalCode uses AST walking to generate an annotated dictionary of function names as keys and their associated nodes as values; thus facilitating code analysis via dynamic filtering of functions, variables or statements using this approach.

Get_func() provides an efficient means of retrieving functions dynamically from a filtered **AST** node, taking as its argument the name of the function to extract. The compile() method takes an AST node's code string and converts it into bytecode. Then it runs that bytecode in the little _restricted_globals sandbox. Doing it this way allows the code to execute dynamically while still giving tools to mess with the functions or peek at what's going on inside.

EvalVisit: We implement some additional actions. EvalVisit can take an AST node representing a function and convert it back into runnable code [22]. It uses the handy unparse() method to turn the node into a Python string while keeping the original structure intact. Then compile() gets called to transform that string into bytecode—essentially a binary version that Python can execute [23]. After that, the code needs somewhere to safely run. EvalVisit provides a little closed-off global namespace for this, called _restricted_globals. When EvalVisit initializes, it grabs the current globals() dict and copies it over so

the code has access to variables. Then it uses the exec() function to actually run the compiled bytecode in that walled-off namespace. This keeps everything isolated from messing with the wider environment. The cool part is any functions defined in the executed code now live in that _restricted_globals space. So EvalVisit can dynamically pick them back out later by name if it is needed to access them again after the initial analysis. In a nutshell: EvalVisit uncompiles nodes, compiles them into bytecode, execs that bytecode in a separate namespace, and reuses anything defined inside there.

Integrated into Website with Backend Implementation: Django is intuitive for rapid web development. Following coding best practices like MVC and DRY principles, it lets programmers build sites quickly while encouraging clean, tidy code. Its strengths stack up - nice built-in user interface features like the admin portal, handy tools for routing URLs, solid security protections against attacks, and a super robust framework for interacting with databases called ORM. For APIs specifically, Django Rest Framework takes all Django's strengths up a notch. By offering pre-built components like viewsets, routers, and serializers for translating data, it makes API development a total breeze. Authentication, permissions, filtering, pagination - it handles all the nitty gritty details out of the box. Schemas and integration codes are automatically generated. Since Rest Framework is designed specifically for APIs while integrating seamlessly with Django, a perfect full-stack solution for smooth web API projects. It really expands on Django's benefits for that domain specifically. The combo enables power through projects faster than ever by handling so much of the boilerplate. It's hard to overstate how much time and headaches that saves developers. By leveraging APIs and automation to smooth out the bumpy code submission and evaluation process, SubmitCodeAPIView makes programming education much simpler to manage. And that means more time focusing on the learning itself rather than the logistics headache. Pretty cool how a few API tools can slim down key parts of the coding literacy movement.

```
DEFINE CLASS SubmitCodeAPIView(APIView):
    SET permission_classes TO [AllowAny]

    DEFINE FUNCTION post(self, request, slug, param, options):
        SET python_code TO request.data.get('code')
        RETURN Response(getResults(python_code, slug, param, options))
```

Figure 4. *Pseudocode for API Backend Implementation.*

SubmitCodeAPIView class offers a smooth way for students to turn in coding assignments. With its permission-free post method, anyone can securely send over their solutions using simple HTTP requests without logging in. Opening up access helps provide an inclusive learning experience. The post takes care of organizing the submissions for easy processing later on. Students just need to package up the Python code in a variable - super straightforward. When their script comes in, post grabs it out of the request data so it's ready for the grading workflow. This links up seamlessly with testing and evaluation tools too. The getResults method returns the submitted code and adds parameters like options and test slugs. This extra context enables running comprehensive assessments matched up to the specifics of the coding challenge. Super important with all the different languages and scenarios out there. Whether it's Python or something else, getResults fires up the automated testing suites to put the code through its paces. Everything gets thoroughly checked against pre-written test cases to provide in-depth feedback. Between hassle-free submitting and robust evaluation, SubmitCodeAPIView makes managing real-world programming assignments much less hair-pulling for both students and instructors. Streamlining these pieces helps modern coding courses handle diverse code more smoothly.

This function double checks that a student's Python code looks accurate and runs properly. It uses custom Python tools to inspect things. It then digs through a collection of pre-made Python test cases matching the details provided, saving any good finds for afterwards. The "try" setting gives students quick feedback when just joking around with code to see what happens. This makes the system chill enough to support loosely testing stuff out along with final grading. All in all, it makes the evaluation process accommodating whether messing about or being serious with the coding.

```
DEFINE FUNCTION getResults(python_code, slug, param, options):
  IF slug EQUALS 'python':
    SET _search TO [item.get('test')
          FOR item IN TEST_CASES_PYTHON IF item.get('param') EQUALS
param]
    IF _search.__len__() > 0:
      SET _search TO _search[0]
      IF options EQUALS 'try':
        SET _search TO _search[:2]
      RETURN evaluate_code(python_code=python_code, testcases=_search)
  RETURN {'err': 'err'}
```

Figure 5. *Pseudocode for Code Evaluation Function.*

Unittest Implementation: The TestAST class is a key part of any testing framework, used to check over important code evaluation parts like EvalCode and EvalVisit that analyze programs on the fly [24]. This paper looks closer at how those components evaluate dynamic code. It also profiles how fast things run during unit testing with software like JMeter to spot any performance issues. When setting up the tests, it can create examples of EvalCode and EvalVisit using predefined input data like INPUT_DATA_NO_ARGS or INPUT_DATA_ARGS. It might set the index_test variable to 10 for the tests to use later. The goal is to really put EvalCode and EvalVisit through their paces to ensure they work right when examining code.

```
DEFINE FUNCTION setUp(self) -> None:
  SET self.eval_code_no_args TO EvalCode(INPUT_DATA_NO_ARGS)
  SET self.eval_code_args TO EvalCode(INPUT_DATA_ARGS)
  SET self.eval_visit_no_args TO EvalVisit(INPUT_DATA_NO_ARGS)
  SET self.eval_visit_args TO EvalVisit(INPUT_DATA_ARGS)
  SET self.index_test TO 10
```

Figure 6. *Pseudocode for Unittest Settings.*

The TestAST class has four-unit tests showing off the dynamic code evaluation features. These tests use stuff like EvalCode's ability to grab and run functions, plus its EvalVisit part to fetch functions that might or might not have arguments. Then the tests can check if those functions did what they were supposed to.

A custom TextTestRunner can be chosen when running tests with TestProgram. This let’s tweak the test running and reporting. Tailored test is setup execution and output, this custom text test runner before using TestProgram at the end to launch CustomTextTestRunner. That way when TestProgram fires off the tests, it will use the custom runner configured instead of the default one. So that custom text test runner option is there for testing to run a certain way or results displayed a certain way that the normal tester doesn't do. It makes sure to select it with TestProgram to CustomTextTestRunner the one in charge of executing and showing the test outcomes.

```
_unit_run = TestProgram(
  testRunner=CustomTextTestRunner,
)
```

Figure 7. *Pseudocode for Unittest Run.*

TestAST class is a key piece of the Python testing framework for checking out dynamic code evaluation. This showcase is all about how TestAST is set up and used to really put the EvalCode and EvalVisit classes through their paces when examining code on the fly. It aims to highlight just how flexible and powerful TestAST is for verifying the dynamic evaluation capabilities thanks to relying on EvalCode and EvalVisit to handle the heavy lifting. The goal is to give a practical example of implementing TestAST to leverage those other classes in order to prove out how robust the overall dynamic code testing system can be. Profiling techniques make testing more accurate while providing developers with better information for making informed decisions regarding its effectiveness and performance of their code.

JTE
HCMUTE
ISSN: 2615-9740

JOURNAL OF TECHNICAL EDUCATION SCIENCE
Ho Chi Minh City University of Technology and Education
Website: https://jte.edu.vn
Email: jte@hcmute.edu.vn

TestAST unique is it uses memory_profiler to measure performance during testing. This allows regular unit testing while also getting super useful data on memory usage and speed when examining code on the fly. That profiling gives key details about memory and time when evaluating code dynamically. In the TestAST tests, each unit test method uses @profile decorators so they get automatically profiled top to bottom for thorough performance analysis right when they run.

```
function run_test_with_no_args_using_eval(strategy):
   index = get_test_index()
   expected_result = generate_expected_result('hello', index)
   actual_result = strategy.get_func(f'func_{index}')()
   assert_equal(actual_result, expected_result)

function run_test_with_args_using_eval(strategy):
   index = get_test_index()
   argument_value = 5
   expected_result = argument_value
   actual_result = strategy.get_func(f'func_{index}')(argument_value)
   assert_equal(actual_result, expected_result)

@profile
function test_func_no_args_using_eval_visit():
   run_test_with_no_args_using_eval(self.eval_visit_no_args)

@profile
function test_func_no_args_using_eval_code():
   run_test_with_no_args_using_eval(self.eval_code_no_args)

@profile
function test_func_args_using_eval_visit():
   run_test_with_args_using_eval(self.eval_visit_args)

@profile
function test_func_args_using_eval_code():
   run_test_with_args_using_eval(self.eval_code_args)
```

Figure 8. *Pseudocode for Unittest Functions for TestAST.*

Developers can get a super detailed look at how their code uses memory by using the memory_profiler module. The TestAST unit test methods have @profile decorators so developers can see the memory usage through the full dynamic code evaluation process. As each decorated test runs, the memory profiler tracks how much memory is being used, giving developers valuable insight into any potential memory leakage or inefficient use issues. Now the dynamic code evaluation parts can maximize memory efficiency to really reach their full potential. The @profile decorator also simplifies profiling the execution time by showing exactly where all the time is being spent in the code, making it easy to catch bottlenecks or optimize certain parts. Doing profiling during unit testing specifically lets developers independently understand the performance impact of dynamic code evaluation, which is super helpful. Each test method with @profile is like a full examination collecting comprehensive data on memory and time. This makes profiling even more useful compared to other options.

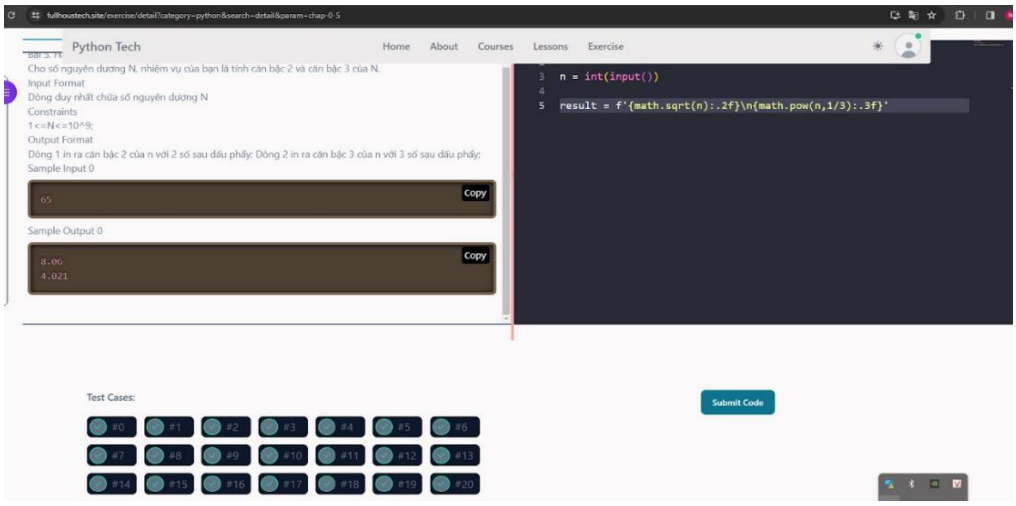

Figure 9. *Interface example of code evaluation system.*

Test Eval Code System into Website (link production: https://fullhoustech.site)

When a user is done coding and clicks submit, tests kick off to see if their solution stacks up to the exercise's needs.

The tests check that the code matches what the problem wanted. Hitting submit starts the automated validation process with those tests examining the user's answers. If everything passes, sweet - their code nailed it. But any fail means their solution didn't fully handle some piece, so they instantly know what still needs work to handle those cases. Behind the scenes, the tests take the submitted code and run it against predetermined use cases to audit the logic. So, clicking submit triggers showtime, letting the user see if their coding holds up when the tester suite inspects it to confirm it checked all the right boxes. Some results are illustrated in Figure 9.

5. Conclusion

Abstract Syntax Trees (ASTs) are a killer way to really dig into student code automatically to support evaluation well. Since multi-threaded code checking can involve fancy analysis, ASTs perfectly fit the bill for reliable needs there.

ASTs make analyzing student code way more accurate - catching issues in their submitted exercises or lecture work, grading things, checking efficiency, offering optimizations. ASTs also pull-out student code crazy quick thanks to custom regular expressions that match common patterns like variables or functions. That speed becomes clutch when student code has to pass a ton of test cases showing it solves something. And ASTs help spot compiler errors or missing pieces early, which is huge for letting students smoothly fix stuff as they learn. Also, most languages play nice with ASTs these days, and Python already bakes them in perfectly - the string input is clutch for tracking all code students type during training.

Dropping in ASTs improves grading and assessment big time, like how LeetCode and HackerRank use them to evaluate code well. Building more assessment tools could double down on ASTs' strengths even more, which those sites already leverage decently for accuracy.

Conclusion as demonstrated above, AST significantly improves accuracy, efficiency and error detection skills essential to grading student code by adding it into the evaluation process. To provide more precise evaluation of students' programming abilities and enhance education platforms' assessment methodologies. This paper advocates its wide implementation on educational platforms. Its process mirrors modern assessment methodologies.

Acknowledgments

This work belongs to the project grant No: SV2024-201 funded by Ho Chi Minh City University of Technology and Education, Vietnam

Conflict of Interest

The authors declare no conflict of interest.

REFERENCES

[1] Y. Taniguchi, T. Minematsu, F. Okubo, and A. Shimada, "Visualizing Source-Code Evolution for Understanding Class-Wide Programming Processes," *Sustainability*, vol. 14, p. 8084, 2022, doi: 10.3390/su14138084.
[2] D. Moore, J. Edwards, H. Karimi, R. Khadka, and P. Bodily, "Temporal Abstract Syntax Trees for Understanding Student Coding Thought Process," *2022 Intermountain Engineering, Technology and Computing (IETC)*, Orem, UT, USA, 2022, pp. 1-6, doi: 10.1109/IETC54973.2022.9796943.
[3] V. Agrahari and S. Chimalakonda, "AST[AR] – Towards Using Augmented Reality and Abstract Syntax Trees for Teaching Data Structures To Novice Programmers," doi: 10.1109/ICALT49669.2020.00100.
[4] J. S. F. M. H. Lulian Neamtiu, "Understanding source code evolution using abstract syntax tree matching," doi: 10.1145/1083142.1083143.
[5] D. Hovemeyer, A. Hellas, A. Petersen, and J. Spacco, "Control-Flow-Only Abstract Syntax Trees for Analyzing Students' Programming Progress," doi: 10.1145/2960310.2960326, 10.1109/ICSE.2019.00086.
[6] C. Lin *et al.*, "Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting," doi: 10.1109/ICPC52881.2021.00026.
[7] S. Peacock, L. Deng, J. Dehlinger, and S. Chakraborty, "Automatic Equivalent Mutants Classification Using Abstract Syntax Tree Neural Networks," doi: 10.1109/ICSTW52544.2021.00016.
[8] D. Yang and C. V. L. Aftab Hussain, "From query to usable code: an analysis of stack overflow code snippets," doi: 10.1145/2901739.2901767.
[9] P. Sedlacek and E. Zaitseva, "Software Reliability Model based on Syntax Tree," doi: 10.1109/IDT52577.2021.9497520.
[10] E. Spirin *et al.*, "PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code," doi: 10.1109/MSR52588.2021.00014.

[11] A. Z. Ablahd, "Using Python to Detect Web application vulnerability," *Resmilitaris*, vol.13, no. 2, 2023.
[12] P. Hozhabrierdi, D. F. Hitos, and C. K. Mohan, "Python Source Code De-Anonymization," doi: 10.1109/ICDMW.2018.00011.
[13] C. Ziyi, L. Lu, and S. Qiu, "An Abstract Syntax Tree Encoding Method for Cross-Project Defect Prediction," doi: 10.1109/ACCESS.2019.2953696.
[14] A. Prochnow and J. Yang, "DiffWatch: watch out for the evolving differential testing in deep learning libraries," doi: 10.1145/3510454.3516835.
[15] Y. S. J. W. I. G. H. Junchen Zhao, "GAP-Gen: Guided Automatic Python Code Generation," doi: 10.48550/arXiv.2201.08810.
[16] M. Zheng, X. Pan, and D. Lillis, "CodEX: Source Code Plagiarism Detection," doi: 329513153.
[17] A. Derezinska and K. Hałas, "Improving Mutation Testing Process of Python Programs," doi: 10.1007/978-3-319-18473-9_23.
[18] T. Wang *et al.*, "PyNose: A Test Smell Detector For Python," doi: 10.1109/ASE51524.2021.9678615.
[19] E. M. Arts, "Towards Querying Abstract Syntax Trees for Python Programs," Master Thesis, Department of Computer Science, Eindhoven University of Technology, The Netherlands, 2022.
[20] M. Duracik *et al.*, "Abstract Syntax Tree Based Source Code Antiplagiarism System for Large Projects Set," doi: 10.1109/ACCESS.2020.3026422.
[21] S. Liu *et al.*, "ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking," doi: 10.1109/TSE.2020.3038681.
[22] X. Li and X. J. Zhong, "The Source Code Plagiarism Detection Using AST," doi: 10.1109/IPTC.2010.90.
[23] D. Perez and S. Chiba, "Cross-Language Clone Detection by Learning Over Abstract Syntax Trees," doi: 10.1109/MSR.2019.00078.
[24] I. Dejanović *et al.*, "A Python tool for the implementation of domain-specific languages," doi: 10.1063/1.4992501.

Nguyen Phuong Anh Tu is currently studying at the Ho Chi Minh City University of Technology and Education, Vietnam, major in Information Technology. Email: 21110105@student.hcmute.edu.vn

Hoang Van Dung received the Ph.D. degree from the University of Ulsan, South Korea, in 2014. He was associated and joined as a visiting researcher with the Intelligence Systems Laboratory, University of Ulsan, 2015. He joined the Robotics Laboratory on Artificial Intelligence, Telecom SudParis as a postdoctoral fellow, 2016. He has been serving as an associate professor in computer science, Faculty of Information Technology, Ho Chi Minh City University of Technology and Education, Vietnam. He has published numerous research articles in ISI, Scopus indexed, and high-impact factor journals. He has been actively participating as a member of the societies as IEEE, IEEE Computer, ICROS. His research interests include a wide area, which focuses on pattern recognition, machine learning, medical image processing, computer vision application, vision-based robotics, and ambient intelligence. Email: dunghv@hcmute.edu.vn. ORCID: https://orcid.org/0000-0001-7554-1707