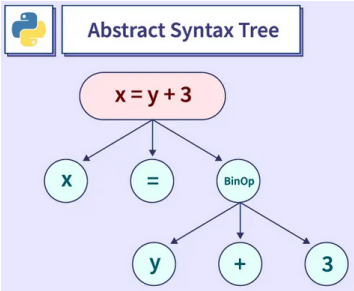
CSC4001 Project: Program Manipulation with AST

Jiayi Yao 120040070@link.cuhk.edu.cn

Due: April 30th

1 Introduction

In this project, you are required to perform various program manipulations with Python abstract syntax tree (AST). Basically, an AST is a tree expression of the program's syntax structure. Figure below illustrates the correspondence between a very naive program and its abstract syntax tree.



An Example of Python AST

In this project, you do not have to worry about the implementation of AST construction because AST is a part of the Python Standard Library. You can simply use the **ast** with **import ast** command. To complete this project, you MUST take advantage of the documentations!!! Here, I provide two commonly-used documentations: (1) Official AST documentation, (2) Green Tree Snakes (HIGHLY recommended). They should be enough for your project, but you can always find more and better information about AST on Google/Youtube.

2 An example to get started

In this section, I'll provide a very specific example to help you better understand how to work with Python AST. However, you still have to read the documents by YOURSELF to tackle this project.

To begin with, you have to import the ast library:

```
import ast
```

To parse the source code into AST, you need the function **ast.parse()**:

```
source = '''
x = 6.5 + 2
y = x + 1
'''

tree = ast.parse(source)
```

To visualize the ast, you can use **ast.dump()**:

```
print(ast.dump(tree, indent=4))
```

The following code snippets illustrate two different ways to replace "1" with "2" using **ast.NodeTransformer**:

```
class Transformer(ast.NodeTransformer):
    def visit_Num(self, node):
        if isinstance(node.n, int) and node.n==1:
            node.n = 2
        return node

Transformer1 = Transformer()
Transformer1.visit(tree)
```

```
class Transformer(ast.NodeTransformer):
    def visit_Constant(self, node):
        if isinstance(node.value, int) and node.value==1:
            node.n = 2
        return node

Transformer1 = Transformer()
Transformer1.visit(tree)
```

You can transform the AST back to source code with **ast.unparse**:

```
print(ast.unparse(tree))
```

3 Questions

In this project, you only have to consider programs with the following components:

- **Data Types:** int, float, boolean
- **Binary Operators:** +, -, *, /
- **Comparison Operators:** ==, >=, <=, >, <
- **Precedence Operators:** (,)
- **Assignment Statement:** e.g., x = 5.7, y = x+True, z = x+5+y
- **Function Definition with no return:** e.g., def foo(a), def foo(b=5), def foo(a, b=5)
- **Function Calls with no assignment:** e.g., foo(5), foo(b=6), foo(a, 5)

The term "no return" means that the function will have no return statement. The term "no assignment" means that there will be no assignment with a function, e.g., x = foo(a). Furthermore, all arguments are defined explicitly. This is to say that special parameter-passing syntax like *args/**kwargs is not included.

For each question, you should include an **input()** function and print out the expected output. Furthermore, you are **ONLY** allowed to use the **ast** library.

3.1 Question 1: Code Mutation (30%)

3.1.1 Problem background

Code mutation is fundamental to many of the software analysis/testing techniques. For example, as shown in the below figure, suppose you have implemented a function **absolute** that returns the absolute value of the input.

```
def absolute(x):
    if x < 0:
        x = -x
    return x
```

To verify the correctness of the program, you have written a test **assertEqual(1, absolute(-1))**. The test passes, but how do you assess the efficacy of your test? One way is to utilize mutation testing which assesses test suite efficacy by inserting small faults into programs and measuring the ability of the test suite to detect them. In this particular scenario, we can perform a very naive mutation by replacing 0 with 1 as shown in the below figure.

```
def absolute(x):
    if x < 1:
        x = -x
    return x
```

This time, the test **assertEqual(1, absolute(-1))** still passes but fails to detect the injected fault. You can also think about how mutation testing can complement code coverage in terms of measuring the effectiveness of test suites.

3.1.2 Problem Statement

In this problem, you should perform the following basic code mutation strategies:

- **Negate Binary Operators:** $+ \Leftrightarrow -$, $* \Leftrightarrow /$
- **Negate Comparison Operators:** $>= \Leftrightarrow <$, $<= \Leftrightarrow >$
- **Delete Unused Function Definitions**

In addition to performing the above mutations, you should also print all the global variables (outside function definitions) one by one with function **print()** at the end of the program. You do not have to consider the syntactic/semantic correctness of the original programs and the mutated programs.

Note that you will not receive any points if you simply treat the program as a string and mutate the string with functions such as **find()** and **replace()**. You should use the ast to perform the mutations. You can think about why mutations with naive string replacement is not a good idea in general.

3.1.3 Input/Output Format

The input is a string representing a Python code snippet. In the output, you should print out the mutated program in the console. Empty lines or spaces between operators are allowed as long as they do not affect the syntax/semantics of the program. For example, expressions **x + 5** and **x+5** are both acceptable.

3.1.4 Examples

```
x = 1
def foo1(x):
    y = x - 5
foo1()
```

Example Input 1

```
x = 1
def foo1(x):
    y = x + 5
foo1()
print(x)
```

Example Output 1

```
x = 1
def foo1():
    x = 1
def foo2():
    y = 6 + (3>=True)
foo2()
```

Example Input 2

```
x = 1
def foo2():
    y = 6 - (3<True)
foo2()
print(x)
```

Example Output 2

3.2 Question 2: Undefined Variables Identification (70%)

3.2.1 Problem Background

An undefined variable is a variable that is accessed in the code but has not been declared by that code. Static undefined variable identification (without running the program) is very useful and has already been applied in many applications. For example, the Python extension in Visual Studio Code marks the undefined variables before the program is actually run. However, debugging can be very time-consuming without undefined variable identification. Let's say when you are doing a homework, you have written a Python code to perform some heavy computations and save the final result to a file. Let's further assume there is only two hours left to ddl, and the program takes an hour and a half to finish. Unluckily, you accidentally introduced an undefined variable somewhere before the saving operation. Once you click the run button, you will certainly miss the ddl.

3.2.2 Problem Statement

In this problem, you are required to count the number of uses of undefined variables. The examples given in section 3.2.4 may help you further understand and clarify the definitions of undefined variable uses. Again, you don't need to worry about the syntactic/semantic correctness of the program (e.g., infinite recursion).

3.2.3 Input/Output Format

The input is a string representing a Python code snippet. The output is simply an integer indicating the number of undefined variable uses.

3.2.4 Examples

In Example 1, the number of undefined variable uses should be 1 since variable y is undefined and used once.

```
x = 1
x = y + 2
```

Example Input 1

```
1
```

Example Output 1

In Example 2, argument y in function call $foo(y)$ is an undefined. However, the number of undefined variable uses would still be 0 since y is never used in the function body.

```
x = 1
def foo(y):
    x = 6
foo(y)
```

```
0
```

Example Input 2 Example Output 2

In Example 3, the undefined variable y will make the variable x in assignment $x = y + 2$ become undefined. And variable x is used later in assignment $z = x + 1$. Thus, the total number of undefined variable uses is 2.

```
x = 1
x = y + 2
z = x + 1
```

```
2
```

Example Input 3 Example Output 3

In example 4, the three undefined variable uses are from: (1) variable y in assignment $x = y + 2$; (2) variable z in assignment $x = y + z$; (3) variable x in assignment $x = x + 1$.

```
x = 1
x = y + 2
def foo(x):
    x = x + 1
def func(z, x, y=6):
    x = y + z
    foo(x)
func(x, 4, y=5)
```

```
3
```

Example Input 4 Example Output 4

4 Requirements

4.1 Language

You should write your code in Python. And as mentioned above, you are only allowed to import the **ast** library.

4.2 Grading

- **Pretests (70%).** For each question, seven pretests are given. If you are able to pass these pretests, you will get at least 70% of the grade. You can find the pretests on blackboard.
- **Hidden tests (30%).** For each question, three additional tests are hidden. They should be very similar to the pretests but longer in code length. And there might be some corner cases which are not included in the pretests.

4.3 Submission

You are only required to upload your source code. No report is required. Name your source code as **1.py** and **2.py** for Question 1 and 2, respectively. Furthermore, you should compress them into a single zip file and name it as **StudentID_proj.zip**. Any **naming error** or **late submission** will not receive any grades.