COSC4315: A Basic Interpreter of Python Programs: Detecting Mutation, Nested if/else, and Recursion Termination

1 Introduction

You will create a Python interpreter written in C++ that can detect variable mutation, nested control statements and linear recursion. You do not have to evaluate the program or consider infinite integers and real numbers.

The input Python program will have simple variables (no lists or arrays) with simple assignment, function definitions (perhaps recursive), if/else control statement (nested), function calls (including recursive), following Python syntax and evaluation semantics. For this homework consider functions with parameters.

2 Input and output

The input is a regular source code Python file (e.g. example.py).

Input example 1: Example of detecting variable mutation

```
a = 5
b = 10

if b > 2:
   c = 50

a = 5 + 3
```

Output example 1

```
Mutated variable: a
Nested if/else level: 0 level
```

Input example 2: Example of detecting nested if/else

```
#Simple program
x = 2
y = 5
z = 7

if x > 0:
   if y > 1:
      if z > 5:
         print(``All three conditions statisfied'')
      else:
         print(``2 conditions satisfied'')
   else:
      print(``1 condition satisfied'')
else:
   print(``No condition satisfied'')
```

Output example 2

```
Nested if/else level: 2 level
```

Input example 3: Example of detecting recursive function

```
def f():
   print(``recursive function'')
   return f()

def f1(n):
   if n==1:
      return 1
   else:
      return n * f1(n-1)

f()
f1(5)
```

Output example 3

```
Recursive function ends: No, Yes
```

Input example 4: Incorrect program. else without if

```
#Incorrect program. else without if
a = 5
b = 3
c = 2

if a>3:
   print("1 condition satisfied")
else:
   print("nothing satisfied")
else:
   print("nothing satisfied")
```

Output example 4

```
Error! else without if.
```

Input example 5: multiple nested if statements

```
a = 5
b = 3
c = 2

if a>3:
   if b<10:
      print("2 conditions satisfied")

if c == 2:
   print("value of c is 2")
else:
   print("value of c is not 2")
```

Output example 5

```
Nested if/else level: 1 level, 0 level
```

Input example 6: Similar to Ackermann function

```
def f(m,n):
   if m==0:
      return n+1
   else:
      return f(m-1,n+1)
f(3,4)
```

Output example 6

```
Recursive function ends: Yes
```

Input example 7: Local variable hiding global variable ¿ no mutation

```
x = 5
z = 4
def f(y):
   x = y + 5
   return x
z = 4 * 20
f(5)
```

Output example 7

```
Mutated variable: z
```

3 Program input and output specification, main call

You will be given a python program (file.py) as input. You have to detect if the program has any mutated variables, how many nested if/else level and if the recursive function ends or not.

The main program should be called **mypython**. The output should be written to the console without any extra characters, but the TAs can redirect it to create some output file. Call syntax at the OS prompt:

```
mypython <file.py>
```

3.1 Output Format:

This is the output format. If the program is missing any functionality such as there is no if/else statement, then you only print the other two.

```
Mutated variable: name of the variables with comma separated. [ex- a, b, c]
Nested if/else level: 0/1/2 level (If multiple if/else then list all the
 levels with comma separated) [ex -0 level, 2 level]
Recursive function ends: Yes/No
```

4 Requirements

- **Detect the mutated variables**. A mutated variable is the one whose value is altered after it is defined. You can assume the expression is integer.
- **Detect Nested if/else statements up to 2 levels**. Note that, a basic if/else statement is level 0.
- **Detect if a recursive function stops or not**. For that, you don't have to evaluate the function. For example, a recursive function that does not have any stopping condition or the recursive function is called incrementally, then it may never finish.
- Your interpreter should behave in the same way as Python.
- Only one statement per line. Treat line number as instruction address.
- Only integer numbers as input (no decimals!).
- Simple arithmetic expressions with one operator.
- One variable assignment per line with full expression on that line. Do not break an arithmetic expression into multiple lines as it will complicate your parser and will make testing more difficult.
- Variable and function names using only letters and digits
- Local and global variables. Functions required to update global variables or create local variables if necessary
- Arithmetic operators: $+-$. Parenthesis required for function definition and function calls.
- functions return only one value (avoid tuples)
- Default Python indentation: 3 spaces
- Support comments with #

- No Python data structures (lists, arrays).
- No OOP constructs like class or method calls.
- No advanced features like I/O, threads, multiple files
- Correctness is the most important requirement: TEST your program with many expressions. Your program should not crash or produce exceptions.

5 Evaluation: Python program interpretation requirements

- Use the Python 3.* interpreter as reference. Test your program comparing with the Python interpreter.
- Allocate variables statically (easier) or dynamically (harder). Deallocate them at the end of the program (easier) or with scoping (harder).
- Output with **print()** statement, following default Python format. Notice Python 2 allows print without parenthesis, but Python 3 does not. Avoid including extra output, if not necessary.
- Catch errors at runtime by default (dynamically). You should identify the error in a specific manner when feasible instead of just displaying an error message.
- Show all the output you want in a trace file, similar to log files. This is essential to debug your program at runtime.
- Optional: Catch errors at parse time, even if it means making multiple passes. Do not worry about minimizing number of passes.

6 Programming requirements

- Must work on our Linux server
- Interpreter must be programmed in GNU C++. Using other C++ compilers is feasible, but discouraged since it is difficult to debug source code. TAs will not test your programs with other C++ compilers (please do not insist).
- Modifying the existing Python open source code is not allowed because it defeats the purpose of learning the basics of programming a lanaguage interpreter. Also, includes many more features than those required in this homework. Finally, our language specification simplifies building the interpreter.
- You can use STL or any C++ libaries or you can develop your own C++ classes. You are required to disclose any code you downloaded/copied.
- You can develop your own scanner/parser or you can use lex, yacc, flex, bison, and so on.
- Create a README file with instructions to compile. Makefile encouraged.
- Your program must compile/run from the command line. There must not be any dependency with your IDE.
- Your mypython program should work in interactive mode or reading an input source code file, like the Python interpreter.