Project #4
CpSc 8270: Language Translation
Computer Science Division, Clemson University
Building an AST
Brian Malloy, PhD
December 12, 2017

Due Date:

In order to receive credit for this assignment, your submission must be submitted, using the `web handin` command, by 8 AM, Monday, November 13^{th} of 2017. If you are unable to complete the project by the first due date, you may submit the project within three days after the due date with a ten point deduction.

Project Description:

Most language translators use a scanner and parser to build an abstract syntax tree (AST) to represent the program, where an AST is a pruned parse tree. The AST is frequently annotated with semantic information about the language constructs in the program and this annotated AST is either (1) still called an AST, or (2) sometimes referred to as an abstract semantic graph (ASG). We will refer to it as an AST.

The advantage of an AST representation of the program is that it represents an abstraction of the program, consisting only of those parts that are needed for translation, visualization, or optimization. For example, the clang compiler uses *llvm*, which builds an AST to represent the program and, using the AST, generates an intermediate representation of the program using *llvm IR*. The *llvm* is well documented and there is an excellent tutorial describing the implantation of a small toy language called *kaleidoscope* at: `https://llvm.org/docs/tutorial/`. The author of this tutorial provides details about the design and implementation of the *llvm* ast, including code that will compile and execute. I can provide information and a sample program if you want to try it out.

Similarly, Python also builds an AST representation of the program and makes this available through a module called ast.py. You can find information about the Python ast at:
`https://docs.python.org/2/library/ast.html`.

In this project, you will use mypy to build an AST representation of the language constructs for expressions that appear in the **global namespace** of a Python program. You will then use the AST to interpret the expressions. You should not translate functions (yet).

Project Specification:

1. Design and implement an Abstract Syntax Tree (AST) to represent and interpret your Python code.
2. Your solution should handle *integer* and *float* values and variables, print, assignment, and expressions such as: {x + y, x − y, x ∗ y, x/y, x//y, x%y, x∗∗e, (x), −x, +x}. Don't forget that Python uses floor for integer division, so that -1/2 is -1, and -1/2-1/2 is also -1.
3. To implement assignment you must build a symbol table. In addition to simple assignment to a variable, your solution should also interpret the following additional forms of assignment:
   { x+ = y, x− = y, x∗ = y, x/ = y, x// = y, x% = y }.
4. In all cases, the oracle for correctness is a Python 2.7.2 interpreter; that is, your expressions should evaluate to the same result that a Python 2.7.2 interpreter would produce. However, unlike the Python

interpreter, the only time you are required to print a result is when the keyword print is used; for example, if the user types x the python interpreter would print the value of x – you are not required to do this in your *mypy* solution. Also, the value is essential but the format is not; for example, for the expression 2.5*2, you may print either 5.0 or 5

5. In the directory that contains your working interpreter, place a new directory titled cases that contains test cases that adequately test your interpreter.
6. Write a test harness, test.py, and place it in your project folder so that it runs the test cases in cases.
7. Your code must be well organized, formatted, readable, free of warnings and leaks, and exploit object orientation.

Resources:

To help you get started, I have provided an example in the course repository that builds an AST for a simple calculator program; you can find the example at: `8270Assets-2017/examples/bison/calculator/ast` This example includes a singleton, SymbolTable, that implements a symbol table for integer variables (but not floats). You can use this example as a starting point or develop your own solution.

In the following execution, my solution is shown on the left:

```
rmalloy@whiterun:~/pubgit/8270-2016/projects/5/code/soln$ r
print -1/2
bam> -1
print -1/2-1/2
bam> -1
print -1/2+(-1/2)
bam> -2
print 2**0.5
bam> 1.41421356237
x = -0.5
print 2**x
bam> 0.707106781187
x = 1/2
print 4**x
bam> 1
```

```
malloy@whiterun:~$ python
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> -1/2
-1
>>> -1/2-1/2
-1
>>> -1/2+(-1/2)
-2
>>> 2**0.5
1.4142135623730951
>>> x = -0.5
>>> 2**x
0.7071067811865476
>>> x = 1/2
>>> 4**x
1
>>>
```

Figure 1: **Execution of my** mypy **juxtaposed with Python 2.7.6**.