Homework – Variables and NASM

Goal:

Utilize a symbol table to track variables. Begin writing the back-end of the compiler by implementing functioning NASM code. Additionally, add in the assignment operator and floating point capabilities.

Purposes:

- Previously, parsed input was processed into a binary tree and/or post-order stack. Now you should turn the post-order stack into actual NASM code.
- Compilers aren't very exciting without an operator = assignment. Add this functionality.
- Utilize a symbol table to track a variable's type and where it is stored in the program stack.
- Learn some basic floating point operations.

Instructions:

Your overall task is to accept everything in the file `accept.txt`, and create correct NASM code which creates a correct program.

Your other overall task is to reject everything wrong in the file `reject.txt`.

The assignment adds some new syntax. You will need new productions for the assignment operator =, a print() call, and floating point numbers. If you struggle determining these productions, ask other students. These can be freely shared.

For all lines in `accept.txt`, translate all instructions into ***unoptimized*** NASM output. In other words, consider these lines:

```
num var8 = 3*4
print(var8)
```

That should create NASM code which will ultimately multiply 3 with 4 and store it in a stack space reserved for `var8`. (Previously, you optimized your IR. Unfortunately, if this assignment allowed you to continue to optimize, you could optimize your compiler down to simply print statements. In class I suggested an approach which upon further reflection was a mess, and so that approach won't be used. You must run unoptimized NASM instructions.)

Ensure the print() statement actually prints to the console. Floating point values can be limited to 3 digits after the decimal.

All variables should have a symbol table entry that relates it to an offset on the program stack.

You can let ints be 32 or 64 bit ints. Likewise, floats can be 32 or 64 bits.

Do not worry about adding additional features that the files do not cover. For example, you don't need to worry about casting between floats and ints.

Disallowed:

- You cannot write an ad-hoc compiler. You must retain your LL(1) parser and its productions. You must also retain a binary tree and/or post-order stack.

- You cannot program your code to reject any lines if it has the word ‘bad’ in it. Students who do will receive unspeakable punishments.

Verification:

- Processing the entire accept.txt file should generate a correct NASM file. That NASM file should be compiled and executed with all outputs displayed.
- Processing the entire reject.txt file should generate an error on each rejected line. You don’t need to specify why the line was rejected if you don’t want to.
- Like prior assignments, a simple 30-60 second video demonstrating your program is required.

Please reach out to me if you have any questions!