1. Translate each of the following Python programs into stack machine code. For our stack machine we have the following primitives:

- **push #a**: push the immediate value a.
- **pop**: pop the top value off the stack.
- **load**: pop a, push mem[a].
- **store**: NEXT -> mem[TOP], pop2.
- **sub**: NEXT - TOP -> a, pop2, push a.
- **cmp**: if NEXT > TOP a = 1 else a = 0, pop2, push a.
- **jmp #a**: jump to a
- **jz #a**: jump to a if TOP = 0, pop
- **call**: pop a, push ra on control stack, jump to a
- **return**: pop ra from control stack and return to it
- **print**: pop a, print a.

Assume that `sub` can be replaced by any binary integer operation we choose (add, mul, or div).

Function arguments should be at the top of the stack when each function is called, and return values should be at the top of the stack when a function returns.

Assume your program starts at label 0 (you may place this wherever you wish).

(a) 

```plaintext
a = 3
b = 4
c = a + b
d = b - a
print d
print c
```

**Answer:**

```plaintext
0: push #3, push #0, store
   push #4, push #1, store
   push #0, load, push #1, load, add
   push #1, load, push #0, load, sub
   print
   print
```
(b)  
\[a = 12\]
\[b = 6\]
if \((a < b)\):
    \[c = a / b\]
else:
    \[c = b - a\]
print \(c\)

Answer:

0:  push #3, push #0, store
    push #4, push #1, store
    push #1, load, push #0, load, cmp
    jz 1
    push #0, load, push #1, load, div
    jmp 2
1:  push #1, load, push #0, load, sub
    jmp 2
2:  print

(c)  
def ackermann(m, n):
    if \((m == 0)\):
        return \(n + 1\)
    else if \((m > 0 \text{ and } n == 0)\):
        return \(\text{ackermann}(m - 1, 1)\)
    else if \((m > 0 \text{ and } n > 0)\):
        return \(\text{ackermann}(m - 1, \text{ackermann}(m, n - 1))\)
    else:
        return 0

print \(\text{ackermann}(2, 3)\)

Answer:

1:  push #1, store
    push #0, store
    push #0, load, jz 2
    jmp 3
2:  push #1, load, push #1, add
    return
3:  push #0, load, push #0, cmp
    jz 5
    push #1, load, jz 4
    jmp 5
4:  push #0, load, push #1, sub
    push #1
(d) (Challenge Question) Come up with your own conventions to represent closures, then write stack machine code for the following:

```python
def f(x):
    def g(z):
        if (x > 0 and z > 0):
            return z
        else:
            return x
    return g

a = f(3)
print a(5)
```