Link for Github for questions 1-3,5,7: <https://github.com/SafwanA12/PLCExam2>

1. github
2. github
3. github

All three functions had different runtimes when the method was called. StaticFunction took the least time to run. stackFunction took a little longer to run. heapFunction took the longest to run by going into 10 seconds. The arrays a & b execute 500000 times and hold 1 million integers. It doesn’t take long for these function to allocate memory the issue with heapFunction is that a heap isn’t sorted like a stack and can only be allocated via pointers. Managing the heap and getting references can be bad on time and efficiency unlike stacks which don’t take much time to create. It’s not possible in Java because you can’t create stack dynamic variables (in Java).

1. Refer to Front.java on github
2. Static Scoping:
3. Local axw (sub3); nonlocal bz (sub2) y(sub1)
4. Local axw (sub3); nonlocal yz (sub1)
5. Local ayz (sub1); nonlocal xw (sub3), b (sub2)
6. Local ayz (sub1); nonlocal xw (sub3)
7. Local abz (sub2); nonlocal xw (sub3), y (sub1)
8. Local ayz(sub1); nonlocal xw(sub3),b (sub2)

Dynamic Scoping:

a)

sub1 y

sub2 b,z

sub3 a,x,w

b)

sub1 y,z

sub3 x,w

c)

sub2 b

sub3 x,w

sub1 a,y,z

1. github

a > b check the values and if a value is greater than b value then it returns true and then checks the values of b and c and compares them with the greater than symbol ‘b > c’ if b value is greater than c the function returns true, next the && operator checks and returns True after returning true the code will evaluate the printf in the if block this prints “True”. Otherwise the code would stop there if returned false. The > operator reads left to right, so the operation executes as ((a > b) > c). Because there is no boolean type for c it uses it. So a > b returns 0 of int. then checks 0 < c so it is false. One true and one false then && operator gives the output 0. So when the ‘if’ condition is not met the code moves on to the else statement printing out false.

* 1. a \* b – 1 + c -> (((a \* b)1)2) + c)3
  2. ++a \* (b – 1) / c % d -> ((((++a)1 \* (b - 1)2)3 / c)4 % d)5
  3. (a – b / c & \* (d \* e / a – 3) -> (((a – b)1 / c)2 & ((d \* e)3 / a)4 – 3)5)6
  4. -a or c = d and e -> ((-a)1 or ((c = d)2 and e)3)4
  5. a > b xor c or d <= 17 -> (((a > b) 1 xor c)3 or (d <= 17)2)4

<expression -> <expression> or<or\_exp>

|<expression>xor<or\_exp>

|<or\_exp>

<or\_exp> -> <or\_exp> and <and\_exp>

|<and\_exp>

<and\_exp> -> <and\_exp> = <exp>

| <and\_exp> /= <exp>

| <and\_exp> < <exp>

| <and\_exp> > <exp>

| <and\_exp> <= <exp>

| <and\_exp> >= <exp>

| <expr>

<expr> -> -> <unary\_expr>

| <unary\_expr>

<unary\_expr> -> <unary\_expr> + <term>

| <unary\_expr> - <term>

| <unary\_expr> & <term>

| <unary\_expr> % <term>

| <term>

<term> -> <term> \* < term>

| <term> / <term.

| not <factor>

| <factor>

<factor> -> (<expr>)

| <operand>

<operand -> a|b|c|d|e