Project 03 - execute part02

Graded

23 Hours, 45 Minutes Late

Student

Ishan Mukherjee

Total Points

100 / 100 pts

Autograder Score
80.0 / 80.0

Passed Tests

Test 0: test03.py
Test 1: test01.py
Test 2: test02.py
Test 3: test03.py
Test 4: test04.py
Test 5: test05.py
Test 6: test06.py
Test 7: test07.py
Test 8: test08.py
Test 9: test09.py
Test 10: test10-24.py
Test 11: test10-24.py
Test 12: test10-24.py
Test 13: test10-24.py
Test 14: test10-24.py
Test 15: test10-24.py
Test 16: test10-24.py
Test 17: test10-24.py
Test 18: test10-24.py
Test 19: test10-24.py
Test 20: test10-24.py
Test 21: test10-24.py
Test 22: test10-24.py
Test 23: test10-24.py
Test 24: test10-24.py
Test 25: test25.py
Test 26: test26.py
Test 27: test27-28.py
Test 28: test27-28.py
Test 29: test29.py
Test 30: test30.py
Test 31: test31.py
Test 32: test32.py
Test 33: test33.py
Test 34: test34-36.py
Test 35: test34-36.py
Test 36: test34-36.py

Question 2

Manual Review

20 / 20 pts

✔ **– 0 pts** Looks good, well done!

– 2 pts "execute.c" header comment has no description

– 2 pts "execute.c" header comment lacking student's name

– 1 pt "execute.c" header comment lacking school, course, etc.

– 5 pts At least one function has no header comment

– 10 pts At least two functions have no header comments

– 15 pts Most / all the functions have no header comments

Autograder Results

Autograder Output

```
**************************************************************************
** Running lizard to analyze coding style, looking to see if functions  **
** exceed 150 lines of code, which is considered too long...           **
**************************************************************************


**************************************************************************
** Lizard analysis tool reports all is well...                        **
**************************************************************************

************************************************************
This is submission #5
Submitted @ 23:44 on 2024-1-27 (Chicago time)

Submission history:
 Submission #4: score=76, submitted @ 23:40 on 2024-1-27 (Chicago time)
 Submission #3: score=79, submitted @ 23:6 on 2024-1-27 (Chicago time)
 Submission #2: score=59, submitted @ 22:54 on 2024-1-27 (Chicago time)
 Submission #1: score=-1, submitted @ 22:12 on 2024-1-27 (Chicago time)

Total # of valid submissions so far: 3
# of valid submissions since midnight: 3
# of minutes since last valid submission: 4
************************************************************
You have 3 submissions this 24-hr period.


****************************************************************
** Number of Submissions This Time Period                     **
****************************************************************

This is submission #4 in current time period

You are allowed a total of 6 submissions per 24-hr time period.

****************************************************************


****************************************************************
** Test Number: 0 **


** Test Input:
print('starting')

x = 456
y = 0.123456789
z = 123.005
```

```
mytrue = True
myfalse = False
a_string_var = "yet another string"
apple = 9102
pass
x = 43.56
y = 87
z = "overwriting with a string"
apple = 1.23498

print('done')


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting
done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 43.560000
 1: y, int, 87
 2: z, str, 'overwriting with a string'
 3: mytrue, boolean, True
 4: myfalse, boolean, False
 5: a_string_var, str, 'yet another string'
 6: apple, real, 1.234980
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 0 **
```

****

****
** Test Number: 1 **

** Test Input:
```
print()
print('starting')
print()

print(123)
print(3.14159)
print(True)
print(False)
print("a really long string that doesn't convey much")

pass

print(0.575)
print("another string")
print(9993312)

print()
print('done')
print()
```

** Your output (first 600 lines) **
```
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...

starting

123
3.141590
True
False
a really long string that doesn't convey much
0.575000
another string
9993312

done
```

```
**done
**MEMORY PRINT**
Capacity: 4
Num values: 0
Contents:
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 1 **
************************************************************************


************************************************************************
** Test Number: 2 **


** Test Input:
print('starting')
pass

x = 456
y = 0.123456789
z = 123.005
mytrue = True
myfalse = False
a_string_var = "yet another string"
apple = 9102

pass
print('done')


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
```

```
**END PRINT**
**executing...
starting
done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, int, 456
 1: y, real, 0.123457
 2: z, real, 123.005000
 3: mytrue, boolean, True
 4: myfalse, boolean, False
 5: a_string_var, str, 'yet another string'
 6: apple, int, 9102
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 2 **
************************************************************************


************************************************************************
** Test Number: 3 **


** Test Input:
print('starting')

x = 456
y = 0.123456789
z = 123.005
mytrue = True
myfalse = False
a_string_var = "yet another string"
apple = 9102
pass
```

```
x = 43.56
y = 87
z = "overwriting with a string"
apple = 1.23498

print('done')


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting
done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 43.560000
 1: y, int, 87
 2: z, str, 'overwriting with a string'
 3: mytrue, boolean, True
 4: myfalse, boolean, False
 5: a_string_var, str, 'yet another string'
 6: apple, real, 1.234980
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 3 **
****************************************************************


****************************************************************
** Test Number: 4 **
```

** Test Input:
```
print('starting yet another test')
print()

x = 456
y = 0.123456789
z = 123.005
print(x)
print(y)
print(z)

mytrue = True
myfalse = False
a_string_var = "yet another string"
apple = 9102

print(apple)
print(a_string_var)
print(mytrue)
print(myfalse)

x = 43.56
y = 87
z = "overwriting with a string"
apple = 1.23498
myfalse = True
print(x)
print(y)
print(z)
print(myfalse)
print(apple)

x = False
print(x)

print()
print('done')
```

** Your output (first 600 lines) **
```
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting yet another test
```

```
456
0.123457
123.005000
9102
yet another string
True
False
43.560000
87
overwriting with a string
True
1.234980
False

done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, boolean, False
 1: y, int, 87
 2: z, str, 'overwriting with a string'
 3: mytrue, boolean, True
 4: myfalse, boolean, True
 5: a_string_var, str, 'yet another string'
 6: apple, real, 1.234980
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 4 **
************************************************************************


************************************************************************
** Test Number: 5 **
```

** Test Input:
```
print('starting yet another test')
print()

x = 456
y = 0.123456789
z = 123.005
print(x)
print(y)
print(z)

mytrue = True
myfalse = False
a_string_var = "yet another string"
apple = 9102

print(apple)
print(a_string_var)
print(mytrue)
print(myfalse)

x = 43.56
y = 87
z = "overwriting with a string"
apple = 1.23498
myfalse = True
print(x)
print(y)
print(z)
print(myfalse)
print(banana)   ## semantic error, oops

print()
print('done')
```

** Your output (first 600 lines) **
```
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting yet another test

456
0.123457
123.005000
```

```
9102
yet another string
True
False
43.560000
87
overwriting with a string
True
**SEMANTIC ERROR: name 'banana' is not defined (line 30)
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 43.560000
 1: y, int, 87
 2: z, str, 'overwriting with a string'
 3: mytrue, boolean, True
 4: myfalse, boolean, True
 5: a_string_var, str, 'yet another string'
 6: apple, real, 1.234980
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 5 **
************************************************************************


************************************************************************
** Test Number: 6 **


** Test Input:
#
# operators with integers
#
x = 100        # 100
y = x - 140    # -40
```

```
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x   # 625

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
print(y)

#
# random comment
#

i = 100
j = 200
b1_1 = 100 < 200
b1_2 = i < 200
b1_3 = 100 < j
```

```
b1_4 = i < j

i = 123
j = 123
b2_1 = 123 != 123
b2_2 = i != 123
b2_3 = 123 != j
b2_4 = i != j

i = 90
j = 90
b3_1 = 90 >= 90
b3_2 = i >= 90
b3_3 = 90 >= j
b3_4 = i >= j

i = 123
j = 123
b4_1 = 123 == 123
b4_2 = i == 123
b4_3 = 123 == j
b4_4 = i == j

i = 100
j = 200
b5_1 = 100 > 200
b5_2 = i > 200
b5_3 = 100 > j
b5_4 = i > j

i = 200
j = 100
b6_1 = 200 > 100
b6_2 = i > 100
b6_3 = 200 > j
b6_4 = i > j

i = 123
j = 12
b7_1 = 123 != 12
b7_2 = i != 12
b7_3 = 123 != j
b7_4 = i != j
b7_5 = i != i

i = 101
j = 100
b8_1 = 101 <= 100
b8_2 = i <= 100
```

```
b8_3 = 101 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12
j = 123
b9_1 = 12 == 123
b9_2 = i == 123
b9_3 = 12 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
test variables:
60
-39
123
533
5
625
42
0
**done
**MEMORY PRINT**
Capacity: 64
Num values: 52
Contents:
 0: x, int, 42
 1: y, int, 0
 2: z, int, 1600
 3: a, int, 180
```

4: test1, int, 60
5: test2, int, -39
6: test3, int, 123
7: test4, int, 533
8: test5, int, 5
9: test6, int, 625
10: divisor, int, 3
11: i, int, 12
12: j, int, 123
13: b1_1, boolean, True
14: b1_2, boolean, True
15: b1_3, boolean, True
16: b1_4, boolean, True
17: b2_1, boolean, False
18: b2_2, boolean, False
19: b2_3, boolean, False
20: b2_4, boolean, False
21: b3_1, boolean, True
22: b3_2, boolean, True
23: b3_3, boolean, True
24: b3_4, boolean, True
25: b4_1, boolean, True
26: b4_2, boolean, True
27: b4_3, boolean, True
28: b4_4, boolean, True
29: b5_1, boolean, False
30: b5_2, boolean, False
31: b5_3, boolean, False
32: b5_4, boolean, False
33: b6_1, boolean, True
34: b6_2, boolean, True
35: b6_3, boolean, True
36: b6_4, boolean, True
37: b7_1, boolean, True
38: b7_2, boolean, True
39: b7_3, boolean, True
40: b7_4, boolean, True
41: b7_5, boolean, False
42: b8_1, boolean, False
43: b8_2, boolean, False
44: b8_3, boolean, False
45: b8_4, boolean, False
46: b8_5, boolean, True
47: b9_1, boolean, False
48: b9_2, boolean, False
49: b9_3, boolean, False
50: b9_4, boolean, False
51: b9_5, boolean, True
END PRINT

**********I**********
** Your program generated the correct outputs, **
** well done! The last step is to run valgrind, **
** which runs your program again to look for   **
** subtle logic and memory errors...   **
****

** Well done, no logic or memory errors! **

** End of Test 6 **
****

****
** Test Number: 7 **

** Test Input:
#
operators with reals
#
x = 100.234
y = x - 140.9
z = y
a = 1.5 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y
print("and now for z:")
print(z)

test1 = x + y
x = 1.2
test2 = x + y
y = 123.123
test3 = x * y
test4 = z / 3.0

```
test5 = z % 11.0
x = 4.2
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3.0
x = 126.9 / divisor
y = 126.9 % divisor
print(x)
print(y)

#
# random comment
#

i = 100.2
j = 200.9
b1_1 = 100.2 < 200.9
b1_2 = i < 200.9
b1_3 = 100.2 < j
b1_4 = i < j

i = 123.0
j = 123.0
b2_1 = 123.0 != 123.0
b2_2 = i != 123.0
b2_3 = 123.0 != j
b2_4 = i != j

i = 90.5
j = 90.5
b3_1 = 90.5 >= 90.5
b3_2 = i >= 90.4
b3_3 = 90.4 >= j
b3_4 = i >= j

i = 123.625
j = 123.625
b4_1 = 123.625 == 123.625
b4_2 = i == 123.5
b4_3 = 123.5 == j
b4_4 = i == j
```

```
i = 100.5
j = 200.5
b5_1 = 100.5 > 200.5
b5_2 = i > 200.5
b5_3 = 100.5 > j
b5_4 = i > j

i = 200.5
j = 100.5
b6_1 = 200.5 > 100.5
b6_2 = i > 100.625
b6_3 = 200.625 > j
b6_4 = i > j

i = 123.123
j = 12.5
b7_1 = 123.123 != 123.5
b7_2 = i != 123.999
b7_3 = 12.5 != j
b7_4 = i != j
b7_5 = j != j

i = 101.25
j = 100.25
b8_1 = 101.25 <= 100.25
b8_2 = i <= 100.25
b8_3 = 100.5 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12.5
j = 123.5
b9_1 = 12.5 == 123.5
b9_2 = i == 123.5
b9_3 = 123.25 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
```

```
100.234000
y is:
-40.666000
z is:
-40.666000
a is:
42.166000
and now for z:
1653.723556
test variables:
59.568000
-39.466000
147.747600
551.241185
3.723556
250.047857
42.300000
0.900000
**done
**MEMORY PRINT**
Capacity: 64
Num values: 52
Contents:
 0: x, real, 42.300000
 1: y, real, 0.900000
 2: z, real, 1653.723556
 3: a, real, 42.166000
 4: test1, real, 59.568000
 5: test2, real, -39.466000
 6: test3, real, 147.747600
 7: test4, real, 551.241185
 8: test5, real, 3.723556
 9: test6, real, 250.047857
 10: divisor, real, 3.000000
 11: i, real, 12.500000
 12: j, real, 123.500000
 13: b1_1, boolean, True
 14: b1_2, boolean, True
 15: b1_3, boolean, True
 16: b1_4, boolean, True
 17: b2_1, boolean, False
 18: b2_2, boolean, False
 19: b2_3, boolean, False
 20: b2_4, boolean, False
 21: b3_1, boolean, True
 22: b3_2, boolean, True
 23: b3_3, boolean, False
 24: b3_4, boolean, True
 25: b4_1, boolean, True
```

```
 26: b4_2, boolean, False
 27: b4_3, boolean, False
 28: b4_4, boolean, True
 29: b5_1, boolean, False
 30: b5_2, boolean, False
 31: b5_3, boolean, False
 32: b5_4, boolean, False
 33: b6_1, boolean, True
 34: b6_2, boolean, True
 35: b6_3, boolean, True
 36: b6_4, boolean, True
 37: b7_1, boolean, True
 38: b7_2, boolean, True
 39: b7_3, boolean, False
 40: b7_4, boolean, True
 41: b7_5, boolean, False
 42: b8_1, boolean, False
 43: b8_2, boolean, False
 44: b8_3, boolean, False
 45: b8_4, boolean, False
 46: b8_5, boolean, True
 47: b9_1, boolean, False
 48: b9_2, boolean, False
 49: b9_3, boolean, False
 50: b9_4, boolean, False
 51: b9_5, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 7 **
****************************************************************


****************************************************************
** Test Number: 8 **


** Test Input:
```

```
#
# operators with mix of ints and reals
#
x = 100
y = x - 140.9
z = y
a = 1 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = a * x
print("and now for z:")
print(z)

test1 = x + y
x = 1
test2 = y + x
y = 123.123
test3 = x * y
test4 = z / 3
x = 1000
test5 = z % 11.1
x = 4
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126.5 / divisor
y = 126.5 % divisor
print(x)
print(y)

#
# random comment
#
```

```
i = 100
j = 200.9
b1_1 = 100 < 200.9
b1_2 = i < 200.9
b1_3 = 100 < j
b1_4 = i < j

i = 123.0
j = 123
b2_1 = 123.0 != 123
b2_2 = i != 123
b2_3 = 123.0 != j
b2_4 = i != j

i = 90.5
j = 90
b3_1 = 90.5 >= 90
b3_2 = i >= 90
b3_3 = 90.4 >= j
b3_4 = i >= j

i = 123.625
j = 124
b4_1 = 123.625 == 124
b4_2 = i == 124
b4_3 = 123.5 == j
b4_4 = i == j

i = 100.5
j = 200
b5_1 = 100.5 > 200
b5_2 = i > 200
b5_3 = 100.5 > j
b5_4 = i > j

i = 200
j = 100.5
b6_1 = 200 > 100.5
b6_2 = i > 100.625
b6_3 = 200 > j
b6_4 = i > j

i = 123.123
j = 12
b7_1 = 123.123 != 12
b7_2 = i != 12
b7_3 = 12.5 != j
b7_4 = i != j
```

```
b7_5 = j != j

i = 101.25
j = 100.25
b8_1 = 101 <= 100.25
b8_2 = i <= 100.25
b8_3 = 100 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12.5
j = 123
b9_1 = 12.5 == 123
b9_2 = i == 123
b9_3 = 123.25 == j
b9_4 = i == j
b9_5 = i == i




** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40.900000
z is:
-40.900000
a is:
41.900000
and now for z:
4190.000000
test variables:
59.100000
-39.900000
123.123000
1396.666667
5.300000
789.048100
42.166667
0.500000
**done
**MEMORY PRINT**
```

Capacity: 64
Num values: 52
Contents:
0: x, real, 42.166667
1: y, real, 0.500000
2: z, real, 4190.000000
3: a, real, 41.900000
4: test1, real, 59.100000
5: test2, real, -39.900000
6: test3, real, 123.123000
7: test4, real, 1396.666667
8: test5, real, 5.300000
9: test6, real, 789.048100
10: divisor, int, 3
11: i, real, 12.500000
12: j, int, 123
13: b1_1, boolean, True
14: b1_2, boolean, True
15: b1_3, boolean, True
16: b1_4, boolean, True
17: b2_1, boolean, False
18: b2_2, boolean, False
19: b2_3, boolean, False
20: b2_4, boolean, False
21: b3_1, boolean, True
22: b3_2, boolean, True
23: b3_3, boolean, True
24: b3_4, boolean, True
25: b4_1, boolean, False
26: b4_2, boolean, False
27: b4_3, boolean, False
28: b4_4, boolean, False
29: b5_1, boolean, False
30: b5_2, boolean, False
31: b5_3, boolean, False
32: b5_4, boolean, False
33: b6_1, boolean, True
34: b6_2, boolean, True
35: b6_3, boolean, True
36: b6_4, boolean, True
37: b7_1, boolean, True
38: b7_2, boolean, True
39: b7_3, boolean, True
40: b7_4, boolean, True
41: b7_5, boolean, False
42: b8_1, boolean, False
43: b8_2, boolean, False
44: b8_3, boolean, True
45: b8_4, boolean, False

```
 46: b8_5, boolean, True
 47: b9_1, boolean, False
 48: b9_2, boolean, False
 49: b9_3, boolean, False
 50: b9_4, boolean, False
 51: b9_5, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 8 **
************************************************************************


************************************************************************
** Test Number: 9 **


** Test Input:
#
# int, real, string concat
#
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
```

```
d = x + 1
some_var = "cs"

print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
print(e)

f = some_var + " 211"
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "

e = "211"
print("")

var1 = x + a
var2 = b + y
var3 = c + e
var4 = d + z
var5 = b + x
var6 = a + y

x = x + x
b = b + b
e = e + e

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)
print(var6)
print(x)
print(b)
print(e)

s1 = "apple"
s2 = "APPLE"
s3 = "banana"
```

```
s4 = "pear"
s5 = "banana"

b1_1 = s1 == s2
b1_2 = s1 == "APPLE"
b1_3 = s1 == "apple"
b1_4 = s1 == "applesauce"
b1_5 = "APPLE" == s2
b1_6 = "APPLE" == s1
b1_7 = s3 == s5
b1_8 = s3 == s4
b1_9 = s3 == s3

b2_1 = s1 != s2
b2_2 = s1 != "APPLE"
b2_3 = s1 != "apple"
b2_4 = s1 != "applesauce"
b2_5 = "APPLE" != s2
b2_6 = "APPLE" != s1
b2_7 = s3 != s5
b2_8 = s3 != s4
b2_9 = s3 != s3

b3_1 = s1 < s2
b3_2 = s1 < "APPLE"
b3_3 = s1 < "apple"
b3_4 = s1 < "applesauce"
b3_5 = "APPLE" < s2
b3_6 = "APPLE" < s1
b3_7 = s3 < s5
b3_8 = s3 < s4
b3_9 = s3 < s3

b4_1 = s1 > s2
b4_2 = s1 > "APPLE"
b4_3 = s1 > "apple"
b4_4 = s1 > "applesauce"
b4_5 = "APPLE" > s2
b4_6 = "APPLE" > s1
b4_7 = s3 > s5
b4_8 = s3 > s4
b4_9 = s3 > s3

b5_1 = s1 <= s2
b5_2 = s1 <= "APPLE"
b5_3 = s1 <= "apple"
b5_4 = s1 <= "applesauce"
b5_5 = "APPLE" <= s2
b5_6 = "APPLE" <= s1
```

```
b5_7 = s3 <= s5
b5_8 = s3 <= s4
b5_9 = s3 <= s3

b6_1 = s1 >= s2
b6_2 = s1 >= "APPLE"
b6_3 = s1 >= "apple"
b6_4 = s1 >= "applesauce"
b6_5 = "APPLE" >= s2
b6_6 = "APPLE" >= s1
b6_7 = s3 >= s5
b6_8 = s3 >= s4
b6_9 = s3 >= s3

print("")
print("done")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
cs 211

11
14.175000
cs 211
a very long string of word that could be many many words --- did you dynamically allocate? shorter
4.675000
20.500000
2
7.350000
211211

done
**done
```

MEMORY PRINT
Capacity: 128
Num values: 75
Contents:
 0: x, int, 2
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, int, 10
 4: b, real, 7.350000
 5: c, str, 'cs '
 6: d, str, 'a very long string of word that could be many many words --- did you dynamically allocate? '
 7: some_var, str, 'cs'
 8: e, str, '211211'
 9: f, str, 'cs 211'
 10: var1, int, 11
 11: var2, real, 14.175000
 12: var3, str, 'cs 211'
 13: var4, str, 'a very long string of word that could be many many words --- did you dynamically allocate? sho
 14: var5, real, 4.675000
 15: var6, real, 20.500000
 16: s1, str, 'apple'
 17: s2, str, 'APPLE'
 18: s3, str, 'banana'
 19: s4, str, 'pear'
 20: s5, str, 'banana'
 21: b1_1, boolean, False
 22: b1_2, boolean, False
 23: b1_3, boolean, True
 24: b1_4, boolean, False
 25: b1_5, boolean, True
 26: b1_6, boolean, False
 27: b1_7, boolean, True
 28: b1_8, boolean, False
 29: b1_9, boolean, True
 30: b2_1, boolean, True
 31: b2_2, boolean, True
 32: b2_3, boolean, False
 33: b2_4, boolean, True
 34: b2_5, boolean, False
 35: b2_6, boolean, True
 36: b2_7, boolean, False
 37: b2_8, boolean, True
 38: b2_9, boolean, False
 39: b3_1, boolean, False
 40: b3_2, boolean, False
 41: b3_3, boolean, False
 42: b3_4, boolean, True
 43: b3_5, boolean, False
 44: b3_6, boolean, True

```
 45: b3_7, boolean, False
 46: b3_8, boolean, True
 47: b3_9, boolean, False
 48: b4_1, boolean, True
 49: b4_2, boolean, True
 50: b4_3, boolean, False
 51: b4_4, boolean, False
 52: b4_5, boolean, False
 53: b4_6, boolean, False
 54: b4_7, boolean, False
 55: b4_8, boolean, False
 56: b4_9, boolean, False
 57: b5_1, boolean, False
 58: b5_2, boolean, False
 59: b5_3, boolean, True
 60: b5_4, boolean, True
 61: b5_5, boolean, True
 62: b5_6, boolean, True
 63: b5_7, boolean, True
 64: b5_8, boolean, True
 65: b5_9, boolean, True
 66: b6_1, boolean, True
 67: b6_2, boolean, True
 68: b6_3, boolean, True
 69: b6_4, boolean, False
 70: b6_5, boolean, True
 71: b6_6, boolean, False
 72: b6_7, boolean, True
 73: b6_8, boolean, False
 74: b6_9, boolean, True
**END PRINT**


**********I**********
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
**********************************************


** Well done, no logic or memory errors! **


** End of Test 9 **
************************************************************************


************************************************************************
```

** Test Number: 10 **

** Test Input:

```
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x   # 625

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisorrr   ## error: undefined
print(x)
print(y)
```

```
print()
print('done')
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
test variables:
60
-39
123
533
5
625
**SEMANTIC ERROR: name 'divisorrr' is not defined (line 42)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 11
Contents:
 0: x, int, 42
 1: y, int, 123
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
 6: test3, int, 123
 7: test4, int, 533
 8: test5, int, 5
 9: test6, int, 625
 10: divisor, int, 3
**END PRINT**
```

**********I**********
** Your program generated the correct outputs, **
** well done! The last step is to run valgrind, **
** which runs your program again to look for   **
** subtle logic and memory errors...   **
****

** Well done, no logic or memory errors! **

** End of Test 10 **
****

****
** Test Number: 11 **

** Test Input:

```
#
# operators with reals -- semantic error
#
x = 100.234
y = x - 140.9
z = y
a = 1.5 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y
print("and now for z:")
print(z)

test1 = x + y
x = 1.2
test2 = x + y
y = 123.123
test3 = x * y
test4 = xyz / 3.0    ## error: undefined
test5 = z % 11.0
x = 4.2
```

```
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3.0
x = 126.9 / divisor
y = 126.9 % divisor
print(x)
print(y)

print()
print('done')
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100.234000
y is:
-40.666000
z is:
-40.666000
a is:
42.166000
and now for z:
1653.723556
**SEMANTIC ERROR: name 'xyz' is not defined (line 27)
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 1.200000
 1: y, real, 123.123000
 2: z, real, 1653.723556
 3: a, real, 42.166000
```

```
 4: test1, real, 59.568000
 5: test2, real, -39.466000
 6: test3, real, 147.747600
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 11 **
**********************************************************************


**********************************************************************
** Test Number: 12 **


** Test Input:
#
# int, real, string concat
#
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
d = x + 1
some_var = "cs"
```

```
print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
print(e)

f = some_variable + " 211"      ## error: undefined
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "

e = "211"
print("")

var1 = x + a
var2 = b + y
var3 = c + e
var4 = d + z
var5 = b + x
var6 = a + y

x = x + x
b = b + b
e = e + e

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)
print(var6)
print(x)
print(b)
print(e)

print("")
print("done")
```

** Your output (first 600 lines) **

**no syntax errors...

```
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
**SEMANTIC ERROR: name 'some_variable' is not defined (line 31)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 9
Contents:
 0: x, int, 2
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, str, 'shorter string'
 4: b, real, 5.675000
 5: c, real, 20.500000
 6: d, int, 3
 7: some_var, str, 'cs'
 8: e, str, 'shorter+a very long string of word that could be many many words --- did you dynamically allocate?
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 12 **
************************************************************************


************************************************************************
```

** Test Number: 13 **

** Test Input:

```
#
# operators with mix of ints and reals -- semantic error
#
x = 100
y = x - 140.9
z = y
a = 1 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = a * x
print("and now for z:")
print(z)

test1 = x + y
x = 1
test2 = y + x
y = 123.123
test3 = x * y
test4 = z / 3
x = 1000
test5 = z % 11.1
x = 4
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126.5 / divisor
y = 126.5 % divisor
print(x)
print(y)
```

```
z = fred    ## error: undefined

#
# random comment
#

i = 100
j = 200.9
b1_1 = 100 < 200.9
b1_2 = i < 200.9
b1_3 = 100 < j
b1_4 = i < j

print()
print('done')



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40.900000
z is:
-40.900000
a is:
41.900000
and now for z:
4190.000000
test variables:
59.100000
-39.900000
123.123000
1396.666667
5.300000
789.048100
42.166667
0.500000
**SEMANTIC ERROR: name 'fred' is not defined (line 47)
**done
**MEMORY PRINT**
Capacity: 16
```

```
Num values: 11
Contents:
 0: x, real, 42.166667
 1: y, real, 0.500000
 2: z, real, 4190.000000
 3: a, real, 41.900000
 4: test1, real, 59.100000
 5: test2, real, -39.900000
 6: test3, real, 123.123000
 7: test4, real, 1396.666667
 8: test5, real, 5.300000
 9: test6, real, 789.048100
 10: divisor, int, 3
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 13 **
**********************************************************************


**********************************************************************
** Test Number: 14 **


** Test Input:
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
```

```
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x   # 625

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
print(y)

#
# random comment
#

i = 100
j = 200
b1_1 = 100 < 200
b1_2 = i < 200
b1_3 = 100 < j
b1_4 = i < j

i = 123
j = 123
b2_1 = 123 != 123
b2_2 = i != 123
b2_3 = 123 != j
b2_4 = i != j
```

```
i = 90
j = 90
b3_1 = 90 >= 90
b3_2 = i >= 90
b3_3 = 90 >= j
b3_4 = i >= j

i = 123
j = 123
b4_1 = 123 == 123
b4_2 = i == 123
b4_3 = 123 == j
b4_4 = i == j

i = 100
j = 200
b5_1 = 100 > 200
b5_2 = i > 200
b5_3 = 100 > j
b5_4 = i > j

i = 200
j = 100
b6_1 = 200 > 100
b6_2 = i > 100
b6_3 = 200 > j
b6_4 = i > j

i = 123
j = 12
b7_1 = 123 != 12
b7_2 = i != 12
b7_3 = 123 != j
b7_4 = i != j
b7_5 = i != i

i = 101
j = 100
b8_1 = 101 <= 100
b8_2 = abc <= 100          ## error: undefined
b8_3 = 101 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12
j = 123
b9_1 = 12 == 123
b9_2 = i == 123
```

```
b9_3 = 12 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
test variables:
60
-39
123
533
5
625
42
0
**SEMANTIC ERROR: name 'abc' is not defined (line 103)
**done
**MEMORY PRINT**
Capacity: 64
Num values: 43
Contents:
 0: x, int, 42
 1: y, int, 0
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
 6: test3, int, 123
 7: test4, int, 533
 8: test5, int, 5
 9: test6, int, 625
 10: divisor, int, 3
```

```
 11: i, int, 101
 12: j, int, 100
 13: b1_1, boolean, True
 14: b1_2, boolean, True
 15: b1_3, boolean, True
 16: b1_4, boolean, True
 17: b2_1, boolean, False
 18: b2_2, boolean, False
 19: b2_3, boolean, False
 20: b2_4, boolean, False
 21: b3_1, boolean, True
 22: b3_2, boolean, True
 23: b3_3, boolean, True
 24: b3_4, boolean, True
 25: b4_1, boolean, True
 26: b4_2, boolean, True
 27: b4_3, boolean, True
 28: b4_4, boolean, True
 29: b5_1, boolean, False
 30: b5_2, boolean, False
 31: b5_3, boolean, False
 32: b5_4, boolean, False
 33: b6_1, boolean, True
 34: b6_2, boolean, True
 35: b6_3, boolean, True
 36: b6_4, boolean, True
 37: b7_1, boolean, True
 38: b7_2, boolean, True
 39: b7_3, boolean, True
 40: b7_4, boolean, True
 41: b7_5, boolean, False
 42: b8_1, boolean, False
**END PRINT**


**********I**********
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
**********************************************


** Well done, no logic or memory errors! **


** End of Test 14 **
****************************************************************
```

```
**********************************************************************
** Test Number: 15 **


** Test Input:
#
# operators with reals -- semantic error
#
x = 100.234
y = x - 140.9
z = y
a = 1.5 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y
print("and now for z:")
print(z)

test1 = x + y
x = 1.2
test2 = x + y
y = 123.123
test3 = x * y
test4 = z / 3.0
test5 = z % 11.0
x = 4.2
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3.0
x = 126.9 / divisor
y = 126.9 % divisor
print(x)
```

```
print(y)

#
# random comment
#

i = 100.2
j = 200.9
b1_1 = 100.2 < 200.9
b1_2 = i < 200.9
b1_3 = 100.2 < zebra       ## error: undefined
b1_4 = i < j

i = 123.0
j = 123.0
b2_1 = 123.0 != 123.0
b2_2 = i != 123.0
b2_3 = 123.0 != j
b2_4 = i != j

i = 90.5
j = 90.5
b3_1 = 90.5 >= 90.5
b3_2 = i >= 90.4
b3_3 = 90.4 >= j
b3_4 = i >= j

i = 123.625
j = 123.625
b4_1 = 123.625 == 123.625
b4_2 = i == 123.5
b4_3 = 123.5 == j
b4_4 = i == j

i = 100.5
j = 200.5
b5_1 = 100.5 > 200.5
b5_2 = i > 200.5
b5_3 = 100.5 > j
b5_4 = i > j

i = 200.5
j = 100.5
b6_1 = 200.5 > 100.5
b6_2 = i > 100.625
b6_3 = 200.625 > j
b6_4 = i > j

i = 123.123
```

```
j = 12.5
b7_1 = 123.123 != 123.5
b7_2 = i != 123.999
b7_3 = 12.5 != j
b7_4 = i != j
b7_5 = j != j

i = 101.25
j = 100.25
b8_1 = 101.25 <= 100.25
b8_2 = i <= 100.25
b8_3 = 100.5 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12.5
j = 123.5
b9_1 = 12.5 == 123.5
b9_2 = i == 123.5
b9_3 = 123.25 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100.234000
y is:
-40.666000
z is:
-40.666000
a is:
42.166000
and now for z:
1653.723556
test variables:
59.568000
-39.466000
147.747600
551.241185
3.723556
250.047857
```

```
42.300000
0.900000
**SEMANTIC ERROR: name 'zebra' is not defined (line 54)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 15
Contents:
 0: x, real, 42.300000
 1: y, real, 0.900000
 2: z, real, 1653.723556
 3: a, real, 42.166000
 4: test1, real, 59.568000
 5: test2, real, -39.466000
 6: test3, real, 147.747600
 7: test4, real, 551.241185
 8: test5, real, 3.723556
 9: test6, real, 250.047857
 10: divisor, real, 3.000000
 11: i, real, 100.200000
 12: j, real, 200.900000
 13: b1_1, boolean, True
 14: b1_2, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 15 **
************************************************************************


************************************************************************
** Test Number: 16 **


** Test Input:
#
# int, real, string concat -- semantic error
#
```

```
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
d = x + 1
some_var = "cs"

print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
print(e)

f = some_var + " 211"
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "

e = "211"
print("")

var1 = x + a
var2 = b + y
var3 = c + e
var4 = d + z
var5 = b + x
var6 = a + y
```

```
x = x + x
b = b + b
e = e + e

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)
print(var6)
print(x)
print(b)
print(e)

s1 = "apple"
s2 = "APPLE"
s3 = "banana"
s4 = "pear"
s5 = "banana"

b1_1 = s1 == s2
b1_2 = s1 == "APPLE"
b1_3 = s1 == "apple"
b1_4 = s1 == "applesauce"
b1_5 = "APPLE" == s2
b1_6 = "APPLE" == s123       ## error: undefined:
b1_7 = s3 == s5
b1_8 = s3 == s4
b1_9 = s3 == s3

b2_1 = s1 != s2
b2_2 = s1 != "APPLE"
b2_3 = s1 != "apple"
b2_4 = s1 != "applesauce"
b2_5 = "APPLE" != s2
b2_6 = "APPLE" != s1
b2_7 = s3 != s5
b2_8 = s3 != s4
b2_9 = s3 != s3

b3_1 = s1 < s2
b3_2 = s1 < "APPLE"
b3_3 = s1 < "apple"
b3_4 = s1 < "applesauce"
b3_5 = "APPLE" < s2
b3_6 = "APPLE" < s1
b3_7 = s3 < s5
b3_8 = s3 < s4
b3_9 = s3 < s3
```

```
b4_1 = s1 > s2
b4_2 = s1 > "APPLE"
b4_3 = s1 > "apple"
b4_4 = s1 > "applesauce"
b4_5 = "APPLE" > s2
b4_6 = "APPLE" > s1
b4_7 = s3 > s5
b4_8 = s3 > s4
b4_9 = s3 > s3

b5_1 = s1 <= s2
b5_2 = s1 <= "APPLE"
b5_3 = s1 <= "apple"
b5_4 = s1 <= "applesauce"
b5_5 = "APPLE" <= s2
b5_6 = "APPLE" <= s1
b5_7 = s3 <= s5
b5_8 = s3 <= s4
b5_9 = s3 <= s3

b6_1 = s1 >= s2
b6_2 = s1 >= "APPLE"
b6_3 = s1 >= "apple"
b6_4 = s1 >= "applesauce"
b6_5 = "APPLE" >= s2
b6_6 = "APPLE" >= s1
b6_7 = s3 >= s5
b6_8 = s3 >= s4
b6_9 = s3 >= s3

print("")
print("done")


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
```

```
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
cs 211

11
14.175000
cs 211
a very long string of word that could be many many words --- did you dynamically allocate? shorter
4.675000
20.500000
2
7.350000
211211
**SEMANTIC ERROR: name 's123' is not defined (line 78)
**done
**MEMORY PRINT**
Capacity: 32
Num values: 26
Contents:
 0: x, int, 2
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, int, 10
 4: b, real, 7.350000
 5: c, str, 'cs '
 6: d, str, 'a very long string of word that could be many many words --- did you dynamically allocate? '
 7: some_var, str, 'cs'
 8: e, str, '211211'
 9: f, str, 'cs 211'
 10: var1, int, 11
 11: var2, real, 14.175000
 12: var3, str, 'cs 211'
 13: var4, str, 'a very long string of word that could be many many words --- did you dynamically allocate? sho
 14: var5, real, 4.675000
 15: var6, real, 20.500000
 16: s1, str, 'apple'
 17: s2, str, 'APPLE'
 18: s3, str, 'banana'
 19: s4, str, 'pear'
 20: s5, str, 'banana'
 21: b1_1, boolean, False
 22: b1_2, boolean, False
 23: b1_3, boolean, True
 24: b1_4, boolean, False
 25: b1_5, boolean, True
**END PRINT**
```

**********I**********
** Your program generated the correct outputs, **
** well done! The last step is to run valgrind, **
** which runs your program again to look for   **
** subtle logic and memory errors...   **
****

** Well done, no logic or memory errors! **

** End of Test 16 **
****

****
** Test Number: 17 **

** Test Input:

```
#
# int, real, string concat -- semantic error
#
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
d = x + 1
some_var = "cs"

print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
print(e)
```

```
f = some_var + " 211"
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "

e = "211"
print("")

var1 = x + a
var2 = b + y
var3 = c + e
var4 = d + z
var5 = b + x
var6 = a + y

x = x + x
b = b + b
e = e + e

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)
print(var6)
print(x)
print(b)
print(e)

s1 = "apple"
s2 = "APPLE"
s3 = "banana"
s4 = "pear"
s5 = "banana"

b1_1 = s1 == s2
b1_2 = s1 == "APPLE"
b1_3 = s1 == "apple"
b1_4 = s1 == "applesauce"
b1_5 = "APPLE" == s2
b1_6 = "APPLE" == s1
```

```
b1_7 = s3 == s5
b1_8 = s3 == s4
b1_9 = s3 == s3

b2_1 = s1 != s2
b2_2 = s1 != "APPLE"
b2_3 = s1 != "apple"
b2_4 = s1 != "applesauce"
b2_5 = "APPLE" != s2
b2_6 = "APPLE" != s1
b2_7 = s3 != s5
b2_8 = s3 != s4
b2_9 = testing123 != s3      ## error: undefined:

b3_1 = s1 < s2
b3_2 = s1 < "APPLE"
b3_3 = s1 < "apple"
b3_4 = s1 < "applesauce"
b3_5 = "APPLE" < s2
b3_6 = "APPLE" < s1
b3_7 = s3 < s5
b3_8 = s3 < s4
b3_9 = s3 < s3

b4_1 = s1 > s2
b4_2 = s1 > "APPLE"
b4_3 = s1 > "apple"
b4_4 = s1 > "applesauce"
b4_5 = "APPLE" > s2
b4_6 = "APPLE" > s1
b4_7 = s3 > s5
b4_8 = s3 > s4
b4_9 = s3 > s3

b5_1 = s1 <= s2
b5_2 = s1 <= "APPLE"
b5_3 = s1 <= "apple"
b5_4 = s1 <= "applesauce"
b5_5 = "APPLE" <= s2
b5_6 = "APPLE" <= s1
b5_7 = s3 <= s5
b5_8 = s3 <= s4
b5_9 = s3 <= s3

b6_1 = s1 >= s2
b6_2 = s1 >= "APPLE"
b6_3 = s1 >= "apple"
b6_4 = s1 >= "applesauce"
b6_5 = "APPLE" >= s2
```

```
b6_6 = "APPLE" >= s1
b6_7 = s3 >= s5
b6_8 = s3 >= s4
b6_9 = s3 >= s3

print("")
print("done")


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
cs 211

11
14.175000
cs 211
a very long string of word that could be many many words --- did you dynamically allocate? shorter
4.675000
20.500000
2
7.350000
211211
**SEMANTIC ERROR: name 'testing123' is not defined (line 91)
**done
**MEMORY PRINT**
Capacity: 64
Num values: 38
Contents:
 0: x, int, 2
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, int, 10
 4: b, real, 7.350000
 5: c, str, 'cs '
```

```
 6: d, str, 'a very long string of word that could be many many words --- did you dynamically allocate? '
 7: some_var, str, 'cs'
 8: e, str, '211211'
 9: f, str, 'cs 211'
 10: var1, int, 11
 11: var2, real, 14.175000
 12: var3, str, 'cs 211'
 13: var4, str, 'a very long string of word that could be many many words --- did you dynamically allocate? sho
 14: var5, real, 4.675000
 15: var6, real, 20.500000
 16: s1, str, 'apple'
 17: s2, str, 'APPLE'
 18: s3, str, 'banana'
 19: s4, str, 'pear'
 20: s5, str, 'banana'
 21: b1_1, boolean, False
 22: b1_2, boolean, False
 23: b1_3, boolean, True
 24: b1_4, boolean, False
 25: b1_5, boolean, True
 26: b1_6, boolean, False
 27: b1_7, boolean, True
 28: b1_8, boolean, False
 29: b1_9, boolean, True
 30: b2_1, boolean, True
 31: b2_2, boolean, True
 32: b2_3, boolean, False
 33: b2_4, boolean, True
 34: b2_5, boolean, False
 35: b2_6, boolean, True
 36: b2_7, boolean, False
 37: b2_8, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 17 **
****************************************************************
```

```
**********************************************************************
** Test Number: 18 **


** Test Input:
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** "x"   ## error: type error

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
```

```
print(y)

print()
print('done')
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
**SEMANTIC ERROR: invalid operand types (line 30)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 9
Contents:
 0: x, int, 4
 1: y, int, 123
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
 6: test3, int, 123
 7: test4, int, 533
 8: test5, int, 5
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************
```

** Well done, no logic or memory errors! **

** End of Test 18 **
****

****
** Test Number: 19 **

** Test Input:
```
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
test2 = x + y    # -39
y = True
test3 = x * y    # error: types
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
```

```
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
print(y)

print()
print('done')
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
**SEMANTIC ERROR: invalid operand types (line 26)
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
 0: x, int, 1
 1: y, boolean, True
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
```

****

** Well done, no logic or memory errors! **

** End of Test 19 **
****

****
** Test Number: 20 **

** Test Input:
```
#
# operators with mix of ints and reals -- semantic error
#
x = 100
y = x - 140.9
z = y
a = 1 - z

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = a * x
print("and now for z:")
print(z)

test1 = x + y
x = 1
test2 = y + x
y = 123.123
test3 = x * y
test4 = z / 3
x = "1000"
test5 = x % 11.1       ## error: type
x = 4
test6 = test5 ** x

print("test variables:")
print(test1)
```

```
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126.5 / divisor
y = 126.5 % divisor
print(x)
print(y)

print()
print('done')
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40.900000
z is:
-40.900000
a is:
41.900000
and now for z:
4190.000000
**SEMANTIC ERROR: invalid operand types (line 29)
**done
**MEMORY PRINT**
Capacity: 8
Num values: 8
Contents:
 0: x, str, '1000'
 1: y, real, 123.123000
 2: z, real, 4190.000000
 3: a, real, 41.900000
 4: test1, real, 59.100000
 5: test2, real, -39.900000
 6: test3, real, 123.123000
 7: test4, real, 1396.666667
**END PRINT**
```

**********I**********
** Your program generated the correct outputs, **
** well done! The last step is to run valgrind, **
** which runs your program again to look for   **
** subtle logic and memory errors...   **
****

** Well done, no logic or memory errors! **

** End of Test 20 **
****

****
** Test Number: 21 **

** Test Input:

```
#
# int, real, string concat -- semantic error
#
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
d = x + 1
some_var = "cs"

print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
```

```
print(e)

f = some_var + " 211"
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "

e = "211"
print("")

var1 = x + a
var2 = b + y
var3 = c + e
var4 = d + z
var5 = b + x
var6 = c + y     ## error: types

x = x + x
b = b + b
e = e + e

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)
print(var6)
print(x)
print(b)
print(e)


print("")
print("done")


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
```

```
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
cs 211

**SEMANTIC ERROR: invalid operand types (line 51)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 15
Contents:
 0: x, int, 1
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, int, 10
 4: b, real, 3.675000
 5: c, str, 'cs '
 6: d, str, 'a very long string of word that could be many many words --- did you dynamically allocate? '
 7: some_var, str, 'cs'
 8: e, str, '211'
 9: f, str, 'cs 211'
 10: var1, int, 11
 11: var2, real, 14.175000
 12: var3, str, 'cs 211'
 13: var4, str, 'a very long string of word that could be many many words --- did you dynamically allocate? sho
 14: var5, real, 4.675000
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 21 **
```

****

****
** Test Number: 22 **

** Test Input:
```
#
# int, real, string concat -- semantic error
#
print("starting")
print("")

x = 1
y = 10.5
z = "shorter"
x = 2

print(x)
print(y)
print(z)

a = z + " string"
print(a)

b = x + 3.675
c = y + 10
d = x + 1
some_var = "cs"

print(b)
print(c)
print(d)

e = z + "+a very long string of word that could be many many words --- did you dynamically allocate?"
print(e)

f = some_var + " 211"
print(f)

x = 1
y = 10.5
z = "shorter"

a = 10
b = 3.675
c = "cs "
d = "a very long string of word that could be many many words --- did you dynamically allocate? "
```

```
e = "211"
print("")

var1 = True
var2 = False
var3 = var1 + var2    ## error: types

print(var1)
print(var2)
print(var3)

print("")
print("done")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

2
10.500000
shorter
shorter string
5.675000
20.500000
3
shorter+a very long string of word that could be many many words --- did you dynamically allocate?
cs 211

**SEMANTIC ERROR: invalid operand types (line 48)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 12
Contents:
 0: x, int, 1
 1: y, real, 10.500000
 2: z, str, 'shorter'
 3: a, int, 10
 4: b, real, 3.675000
 5: c, str, 'cs '
 6: d, str, 'a very long string of word that could be many many words --- did you dynamically allocate? '
 7: some_var, str, 'cs'
```

```
 8: e, str, '211'
 9: f, str, 'cs 211'
 10: var1, boolean, True
 11: var2, boolean, False
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 22 **
************************************************************************


************************************************************************
** Test Number: 23 **


** Test Input:
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
```

```
x = 1
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x   # 625

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
print(y)

#
# random comment
#

i = 100
j = 200
b1_1 = 100 < 200
b1_2 = i < 200
b1_3 = 100 < j
b1_4 = i < j

i = 123
j = 123
b2_1 = 123 != 123
b2_2 = i != 123
b2_3 = 123 != j
b2_4 = i != j

i = 90
j = True
b3_1 = 90 >= 90
b3_2 = i >= 90
b3_3 = 90 >= j     ## error: types
b3_4 = i >= j

i = 123
```

```
j = 123
b4_1 = 123 == 123
b4_2 = i == 123
b4_3 = 123 == j
b4_4 = i == j

i = 100
j = 200
b5_1 = 100 > 200
b5_2 = i > 200
b5_3 = 100 > j
b5_4 = i > j

i = 200
j = 100
b6_1 = 200 > 100
b6_2 = i > 100
b6_3 = 200 > j
b6_4 = i > j

i = 123
j = 12
b7_1 = 123 != 12
b7_2 = i != 12
b7_3 = 123 != j
b7_4 = i != j
b7_5 = i != i

i = 101
j = 100
b8_1 = 101 <= 100
b8_2 = i <= 100
b8_3 = 101 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12
j = 123
b9_1 = 12 == 123
b9_2 = i == 123
b9_3 = 12 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
```

```
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
test variables:
60
-39
123
533
5
625
42
0
**SEMANTIC ERROR: invalid operand types (line 68)
**done
**MEMORY PRINT**
Capacity: 32
Num values: 23
Contents:
 0: x, int, 42
 1: y, int, 0
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
 6: test3, int, 123
 7: test4, int, 533
 8: test5, int, 5
 9: test6, int, 625
 10: divisor, int, 3
 11: i, int, 90
 12: j, boolean, True
 13: b1_1, boolean, True
 14: b1_2, boolean, True
 15: b1_3, boolean, True
 16: b1_4, boolean, True
 17: b2_1, boolean, False
 18: b2_2, boolean, False
 19: b2_3, boolean, False
```

```
 20: b2_4, boolean, False
 21: b3_1, boolean, True
 22: b3_2, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 23 **
**********************************************************************


**********************************************************************
** Test Number: 24 **


** Test Input:
#
# operators with integers -- semantic error
#
x = 100        # 100
y = x - 140    # -40
z = y          # -40
a = 140 - z    # 180

print("x is:")
print(x)
print('y is:')
print(y)
print('z is:')
print(z)
print('a is:')
print(a)

z = y * y      # 1600
print("and now for z:")
print(z)

test1 = x + y    # 60
x = 1
```

```
test2 = x + y    # -39
y = 123
test3 = x * y    # 123
test4 = z / 3    # 533
test5 = z % 11   # 5
x = 4
test6 = test5 ** x   # 625

print("test variables:")
print(test1)
print(test2)
print(test3)
print(test4)
print(test5)
print(test6)

divisor = 3
x = 126 / divisor
y = 126 % divisor
print(x)
print(y)

#
# random comment
#

i = 100
j = 200
b1_1 = 100 < 200
b1_2 = i < 200
b1_3 = 100 < j
b1_4 = i < j

i = 123.456
j = "123"
b2 = i != j        ## error: types

i = 90
j = 90
b3_1 = 90 >= 90
b3_2 = i >= 90
b3_3 = 90 >= j
b3_4 = i >= j

i = 123
j = 123
b4_1 = 123 == 123
b4_2 = i == 123
b4_3 = 123 == j
```

```
b4_4 = i == j

i = 100
j = 200
b5_1 = 100 > 200
b5_2 = i > 200
b5_3 = 100 > j
b5_4 = i > j

i = 200
j = 100
b6_1 = 200 > 100
b6_2 = i > 100
b6_3 = 200 > j
b6_4 = i > j

i = 123
j = 12
b7_1 = 123 != 12
b7_2 = i != 12
b7_3 = 123 != j
b7_4 = i != j
b7_5 = i != i

i = 101
j = 100
b8_1 = 101 <= 100
b8_2 = i <= 100
b8_3 = 101 <= j
b8_4 = i <= j
b8_5 = i <= i

i = 12
j = 123
b9_1 = 12 == 123
b9_2 = i == 123
b9_3 = 12 == j
b9_4 = i == j
b9_5 = j == j



** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
```

```
x is:
100
y is:
-40
z is:
-40
a is:
180
and now for z:
1600
test variables:
60
-39
123
533
5
625
42
0
**SEMANTIC ERROR: invalid operand types (line 59)
**done
**MEMORY PRINT**
Capacity: 32
Num values: 17
Contents:
 0: x, int, 42
 1: y, int, 0
 2: z, int, 1600
 3: a, int, 180
 4: test1, int, 60
 5: test2, int, -39
 6: test3, int, 123
 7: test4, int, 533
 8: test5, int, 5
 9: test6, int, 625
 10: divisor, int, 3
 11: i, real, 123.456000
 12: j, str, '123'
 13: b1_1, boolean, True
 14: b1_2, boolean, True
 15: b1_3, boolean, True
 16: b1_4, boolean, True
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
```

** subtle logic and memory errors... **
****

** Well done, no logic or memory errors! **

** End of Test 24 **
****

****
** Test Number: 25 **

** Test Python:

```
#
# input, int, float
#
print("Starting")

x = 1
y = 10.5
z = "fruit: "
print(x)
print(y)
print(z)

s = input("enter a number> ")
print()
y = int(s)
y = 2 * y

print(y)
print(s)

s2 = input("another number> ")
print()
x = float(s2)
x = x ** 2.0

print(x)
print(s2)

z2 = input("enter whatever you want> ")
print()
z = z + z2

print(z)
```

```
print("Done")


** Test Keyboard Input:
7891
45.625
apples bananas strawberries


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
Starting
1
10.500000
fruit:
enter a number>
15782
7891
another number>
2081.640625
45.625
enter whatever you want>
fruit: apples bananas strawberries
Done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
 0: x, real, 2081.640625
 1: y, int, 15782
 2: z, str, 'fruit: apples bananas strawberries'
 3: s, str, '7891'
 4: s2, str, '45.625'
 5: z2, str, 'apples bananas strawberries'
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
```

****

** Well done, no logic or memory errors! **

** End of Test 25 **
****

****
** Test Number: 26 **

** Test Python:

```
#
# input, int, float --- with 0 and 0.00 as inputs
#
print("Starting")

x = 1
y = 10.5
z = "fruit: "
print(x)
print(y)
print(z)

s = input("enter a number> ")
print()
y = int(s)
y = 10 + y

print(y)
print(s)

s2 = input("another number> ")
print()
x = float(s2)
x = x - y

print(x)
print(s2)

z2 = input("enter whatever you want> ")
print()
z = z + z2

print(z)
```

```
print("Done")


** Test Keyboard Input:
0
0.00
apples bananas pears strawberries


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
Starting
1
10.500000
fruit:
enter a number>
10
0
another number>
-10.000000
0.00
enter whatever you want>
fruit: apples bananas pears strawberries
Done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
 0: x, real, -10.000000
 1: y, int, 10
 2: z, str, 'fruit: apples bananas pears strawberries'
 3: s, str, '0'
 4: s2, str, '0.00'
 5: z2, str, 'apples bananas pears strawberries'
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************
```

** Well done, no logic or memory errors! **

** End of Test 26 **
****

****
** Test Number: 27 **

** Test Python:
#
input, int, float --- with invalid numeric input
#
print("Starting")

x = 3
y = 10.5
z = " END"
print(x)
print(y)
print(z)

z2 = input("enter some text> ")
print()
z = z2 + z

print(z)

s = input("enter a number> ")
print()
y = int(s)
y = 2 * y

print(y)
print(s)

s = input("enter a number> ")
print()
y = int(s)
y = y + x

print(y)
print(s)

s2 = input("another number> ")

```
print()
x = float(s2)
x = x ** 2.0

print(x)
print(s2)

print("Done")


** Test Keyboard Input:
apples bananas pears strawberries
1234
-99
apple


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
Starting
3
10.500000
 END
enter some text>
apples bananas pears strawberries END
enter a number>
2468
1234
enter a number>
-96
-99
another number>
**SEMANTIC ERROR: invalid string for float() (line 37)
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
 0: x, int, 3
 1: y, int, -96
 2: z, str, 'apples bananas pears strawberries END'
 3: z2, str, 'apples bananas pears strawberries'
 4: s, str, '-99'
 5: s2, str, 'apple'
```

END PRINT

**********I**********
** Your program generated the correct outputs, **
** well done! The last step is to run valgrind, **
** which runs your program again to look for   **
** subtle logic and memory errors...   **
****

** Well done, no logic or memory errors! **

** End of Test 27 **
****

****
** Test Number: 28 **

** Test Python:

```
#
# input, int, float --- with invalid numeric input
#
print("Starting")

x = 3
y = 10.5
z = "## "
z2 = " ##"
print(x)
print(y)
print(z)

z3 = input("enter some text> ")
print()
z = z + z3
z = z + z2

print(z)

s = input("enter a number> ")
print()
y = int(s)
y = 2 * y

print(y)
```

```
print(s)

s = input("enter a number> ")
print()
y = int(s)
y = y + x

print(y)
print(s)

s2 = input("another number> ")
print()
x = float(s2)
x = x ** 2.0

print(x)
print(s2)

print("Done")
```

** Test Keyboard Input:
this is a long string, but in reality not that long in the big scheme of things
-123
apple
3.14

** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
PROGRAM GRAPH PRINT
<<omitted to reduce gradescope output>>
END PRINT
**executing...
Starting
3
10.500000
##
enter some text>
this is a long string, but in reality not that long in the big scheme of things
enter a number>
-246
-123
enter a number>
**SEMANTIC ERROR: invalid string for int() (line 31)
**done
MEMORY PRINT
Capacity: 8

```
Num values: 6
Contents:
 0: x, int, 3
 1: y, int, -246
 2: z, str, '## this is a long string, but in reality not that long in the big scheme of things ##'
 3: z2, str, ' ##'
 4: z3, str, 'this is a long string, but in reality not that long in the big scheme of things'
 5: s, str, 'apple'
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 28 **
**********************************************************************


**********************************************************************
** Test Number: 29 **


** Test Python:
#
# loop with input:
#
s = input('Enter an integer> ')
print()
N = int(s)
result = "project03: "

i = 1
x = 3.14159
print('start of loop')
while i <= N:
{
   letter = input('enter a letter> ')
   print()
   result = result + letter
   print(result)
   x = 0.123456789 + x
```

```
  print(x)
  i = i + 1
}
print('end of loop')
print(i)
```

** Test Keyboard Input:
8
a
c
M
q
T
z
X
B

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
Enter an integer>
start of loop
enter a letter>
project03: a
3.265047
enter a letter>
project03: ac
3.388504
enter a letter>
project03: acM
3.511960
enter a letter>
project03: acMq
3.635417
enter a letter>
project03: acMqT
3.758874
enter a letter>
project03: acMqTz
3.882331
enter a letter>
project03: acMqTzX
4.005788
```

```
enter a letter>
project03: acMqTzXB
4.129244
end of loop
9
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
 0: s, str, '8'
 1: N, int, 8
 2: result, str, 'project03: acMqTzXB'
 3: i, int, 9
 4: x, real, 4.129244
 5: letter, str, 'B'
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 29 **
************************************************************************


************************************************************************
** Test Number: 30 **


** Test Input:
#
# loops
#
print("STARTING")
print()

x = 10

while x <= 22:
{
```

```
  print(x)
  x = x + 1
  print(x)
  x = x + 2
  print(x)
  x = 1 + x
}

print()
print("DONE")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
STARTING

10
11
13
14
15
17
18
19
21
22
23
25

DONE
**done
**MEMORY PRINT**
Capacity: 4
Num values: 1
Contents:
 0: x, int, 26
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
```

```
************************************************


** Well done, no logic or memory errors! **


** End of Test 30 **
************************************************************************


************************************************************************
** Test Number: 31 **


** Test Input:
#
# while loops
#
print("starting")
print("")

print("loop #1:")
x = 100
while 112 > x:
{
  print(x)
  x = x + 1
  x = 2 + x
  pass
  pass
  x = x - 1
  pass
  x = x + 2
  pass
}
print(x)
print()

print("loop #2:")
y = 0 - 2.5
z = 0 - 5.5
while z <= y:
{
  pass
  print(y)
  pass
  y = y - 0.5
}
print(y)
```

```
print()

print("loop #3:")
z = "looping with a string?"
while z != "looping with a string?????????":
{
   z = z + "?"
   print(z)
}
print(z)

print("")
print("done")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

loop #1:
100
104
108
112

loop #2:
-2.500000
-3.000000
-3.500000
-4.000000
-4.500000
-5.000000
-5.500000
-6.000000

loop #3:
looping with a string??
looping with a string???
looping with a string????
looping with a string?????
looping with a string??????
looping with a string???????
looping with a string????????
looping with a string?????????
```

```
looping with a string?????????

done
**done
**MEMORY PRINT**
Capacity: 4
Num values: 3
Contents:
 0: x, int, 112
 1: y, real, -6.000000
 2: z, str, 'looping with a string?????????'
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 31 **
*****************************************************************


*****************************************************************
** Test Number: 32 **


** Test Input:
#
# nested loops
#
print("NESTED LOOPS")
print("")

i = 1

while i != 5:
{
  j = i + 1

  while j <= 7:
  {
    print("j")
```

```
    print(j)
    j = j + 1
  }

  i = i + 1
  print("")

  k = i
  while k > 2:
  {
     print("k")
     print(k)
     k = k - 1
  }

  print("")
}

print(i)
print(j)
print(k)

print("")
print("END")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
NESTED LOOPS

j
2
j
3
j
4
j
5
j
6
j
7
```

```
j
3
j
4
j
5
j
6
j
7

k
3

j
4
j
5
j
6
j
7

k
4
k
3

j
5
j
6
j
7

k
5
k
4
k
3

5
8
2

END
**done
**MEMORY PRINT**
```

```
Capacity: 4
Num values: 3
Contents:
 0: i, int, 5
 1: j, int, 8
 2: k, int, 2
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 32 **
**********************************************************************


**********************************************************************
** Test Number: 33 **


** Test Input:
#
# nested loops
#
print("NESTED LOOPS")
print("")

i = 10
loop_end = i + 89

while i >= 0:
{
  j = i - 2
  print(i)
  while j <= 100:
  {
    k = "apple"
    while k != "APPLE":
    {
      var = 99
      while var != loop_end:
```

```
      {
        print('level 4 should never appear')
      }
      print(k)
      k = "APPLE"
    }
    j = j ** 2
    print(j)
  }
  print()
  i = i - 5
}

print('after loop:')
print(i)
print(j)
print(k)
print(var)
print(loop_end)

print("")
print("END")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
NESTED LOOPS

10
apple
64
apple
4096

5
apple
9
apple
81
apple
6561

0
apple
```

```
4
apple
16
apple
256

after loop:
-5
256
APPLE
99
99

END
**done
**MEMORY PRINT**
Capacity: 8
Num values: 5
Contents:
 0: i, int, -5
 1: loop_end, int, 99
 2: j, int, 256
 3: k, str, 'APPLE'
 4: var, int, 99
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 33 **
************************************************************************


************************************************************************
** Test Number: 34 **


** Test Input:
#
# loops --- semantic error
```

```
#
print("starting")
print("")

s = "string"
var = 123

while var != "string???":    # type error
{
   s = s + "?"
   print(var)
}

print("")
print("done")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

**SEMANTIC ERROR: invalid operand types (line 10)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
Contents:
 0: s, str, 'string'
 1: var, int, 123
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 34 **
```

****

****
** Test Number: 35 **

** Test Input:
```
#
# loop --- semantic error
#
print("starting")
print("")

s = "string"
i = 123

while s != "string???":
{
  s = s + "?"
  print(s)

  while i != 0:  # j is not defined
  {
    i = i + j
  }

  print(s)
}

print("")
print("done")
```

** Your output (first 600 lines) **
```
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

string?
**SEMANTIC ERROR: name 'j' is not defined (line 17)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
```

```
Contents:
 0: s, str, 'string?'
 1: i, int, 123
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 35 **
************************************************************************


************************************************************************
** Test Number: 36 **


** Test Input:
#
# loop --- semantic error
#
print("starting")
print()

i = 100
print(i)

while i != 100:
{
   print("you should not see this")
   j = 123
}

#
# the loop never executes so j is not defined above:
#
print(j)

print()
print("done")
```

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
<<omitted to reduce gradescope output>>
**END PRINT**
**executing...
starting

100
**SEMANTIC ERROR: name 'j' is not defined (line 19)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 1
Contents:
 0: i, int, 100
**END PRINT**


**************************I**********************
** Your program generated the correct outputs,  **
** well done! The last step is to run valgrind, **
** which runs your program again to look for    **
** subtle logic and memory errors...            **
************************************************


** Well done, no logic or memory errors! **


** End of Test 36 **
************************************************************************
```

Excellent, perfect score!

Test 0: test03.py

Test 0: test03.py (your main.c, our execute.c) -- yay, output correct!

Test 1: test01.py

Test 1: test01.py (print) -- yay, output correct!

Test 2: test02.py

Test 2: test02.py (x = literal) -- yay, output correct!

Test 3: test03.py

Test 3: test03.py (print(var)) -- yay, output correct!

Test 4: test04.py

Test 4: test04.py (print(var)) -- yay, output correct!

Test 5: test05.py

Test 5: test05.py (semantic error) -- yay, output correct!

Test 6: test06.py

Test 6: test06.py (operators with ints) -- yay, output correct!

Test 7: test07.py

Test 7: test07.py (operators with reals) -- yay, output correct!

Test 8: test08.py

Test 8: test08.py (operators with mix of reals and ints) -- yay, output correct!

Test 9: test09.py

Test 9: test09.py (ints, reals, strings) -- yay, output correct!

Test 10: test10-24.py

Test 10: test10-24.py (semantic errors) -- yay, output correct!

Test 11: test10-24.py

Test 11: test10-24.py (semantic errors) -- yay, output correct!

Test 12: test10-24.py

Test 12: test10-24.py (semantic errors) -- yay, output correct!

Test 13: test10-24.py

Test 13: test10-24.py (semantic errors) -- yay, output correct!

Test 14: test10-24.py

Test 14: test10-24.py (semantic errors) -- yay, output correct!

Test 15: test10-24.py

Test 15: test10-24.py (semantic errors) -- yay, output correct!

Test 16: test10-24.py

Test 16: test10-24.py (semantic errors) -- yay, output correct!

Test 17: test10-24.py

Test 17: test10-24.py (semantic errors) -- yay, output correct!

Test 18: test10-24.py

Test 18: test10-24.py (semantic errors) -- yay, output correct!

Test 19: test10-24.py

Test 19: test10-24.py (semantic errors) -- yay, output correct!

Test 20: test10-24.py

Test 20: test10-24.py (semantic errors) -- yay, output correct!

Test 21: test10-24.py

Test 21: test10-24.py (semantic errors) -- yay, output correct!

Test 22: test10-24.py

Test 22: test10-24.py (semantic errors) -- yay, output correct!

Test 23: test10-24.py

Test 23: test10-24.py (semantic errors) -- yay, output correct!

Test 24: test10-24.py

Test 24: test10-24.py (semantic errors) -- yay, output correct!

Test 25: test25.py

Test 25: test25.py (input, int, float functions) -- yay, output correct!

Test 26: test26.py

Test 26: test25.py (numeric inputs of 0) -- yay, output correct!

Test 27: test27-28.py

Test 27: test27-28.py (invalid numeric input) -- yay, output correct!

Test 28: test27-28.py

Test 28: test27-28.py (invalid numeric input) -- yay, output correct!

Test 29: test29.py

Test 29: test29.py (while loop with input) -- yay, output correct!

Test 30: test30.py

Test 30: test30.py (while loop) -- yay, output correct!

Test 31: test31.py

Test 31: test31.py (while loops) -- yay, output correct!

Test 32: test32.py

Test 32: test32.py (nested while loops) -- yay, output correct!

Test 33: test33.py

Test 33: test33.py (nested while loops) -- yay, output correct!

Test 34: test34-36.py

Test 34: test34-36.py (loop with semantic error) -- yay, output correct!

Test 35: test34-36.py

Test 35: test34-36.py (loop with semantic error) -- yay, output correct!

Test 36: test34-36.py

Test 36: test34-36.py (loop with semantic error) -- yay, output correct!

Submitted Files

main.c

Download

```c
/*main.c*/

//
// Student Name: Ishan Mukherjee
// Course: CS 211
// Term: Winter 2024
//
// Project 02: main program to parse, build program graph, and
// then execute nuPython program.
//
// Solution by Prof. Joe Hummel
// Northwestern University
// CS 211
//

// to eliminate warnings about stdlib in Visual Studio
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>  // true, false
#include <string.h>   // strcspn

#include "token.h"    // token defs
#include "scanner.h"
#include "parser.h"
#include "programgraph.h"
#include "ram.h"
#include "execute.h"


//
// main
//
// usage: program.exe [filename.py]
//
// If a filename is given, the file is opened and serves as
// input to the scanner. If a filename is not given, then
// input is taken from the keyboard until $ is input.
//
int main(int argc, char* argv[])
{
  FILE* input = NULL;
  bool  keyboardInput = false;

  if (argc < 2) {
```

```
47      //
48      // no args, just the program name:
49      //
50      input = stdin;
51      keyboardInput = true;
52    }
53    else {
54      //
55      // assume 2nd arg is a nuPython file:
56      //
57      char* filename = argv[1];
58
59      input = fopen(filename, "r");
60
61      if (input == NULL) // unable to open:
62      {
63        printf("**ERROR: unable to open input file '%s' for input.\n", filename);
64        return 0;
65      }
66
67      keyboardInput = false;
68    }
69
70    if (keyboardInput)  // prompt the user if appropriate:
71    {
72      printf("nuPython input (enter $ when you're done)>\n");
73    }
74
75    //
76    // call parser to check program syntax:
77    //
78    parser_init();
79
80    struct TokenQueue* tokens = parser_parse(input);
81
82    if (tokens == NULL)
83    {
84      //
85      // program has a syntax error, error msg already output:
86      //
87    }
88    else
89    {
90      //
91      // parsing successful, now build program graph:
92      //
93      printf("**no syntax errors...\n");
94      printf("**building program graph...\n");
95
```

```
96      struct STMT* program = programgraph_build(tokens);
97
98      programgraph_print(program);
99
100     //
101     // now execute the program:
102     //
103     printf("**executing...\n");
104
105     struct RAM* memory = ram_init();
106
107     execute(program, memory);
108
109     printf("**done\n");
110
111     ram_print(memory);
112   }
113
114   //
115   // done:
116   //
117   if (!keyboardInput)
118     fclose(input);
119
120   return 0;
121 }
122
```

execute.c Download

```
1  /*execute.c*/
2
3  //
4  // Student: Ishan Mukherjee
5  // Course: CS 211
6  // Term: Winter 2024
7  //
8  // Executes nuPython program, given as a Program Graph.
9  //
10 // Solution by Prof. Joe Hummel
11 // Northwestern University
12 // CS 211
13 //
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <stdbool.h>  // true, false
18 #include <string.h>
19 #include <assert.h>
20 #include <math.h>
21
22 #include "programgraph.h"
23 #include "ram.h"
24 #include "execute.h"
25
26
27 //
28 // Private functions:
29 //
30
31 //
32 // get_element_value
33 //
34 // Given a basic element of an expression --- an identifier
35 // "x" or some kind of literal like 123 --- the value of
36 // this identifier or literal is returned via the reference
37 // parameter. Returns true if successful, false if not.
38 //
39 // Why would it fail? If the identifier does not exist in
40 // memory. This is a semantic error, and an error message is
41 // output before returning.
42 //
43 static bool get_element_value(struct STMT* stmt, struct RAM* memory, struct ELEMENT* element,
   struct RAM_VALUE* value)
44 {
45   if (element->element_type == ELEMENT_INT_LITERAL) {
```

```
46
47    char* literal = element->element_value;
48
49    value->value_type = RAM_TYPE_INT;
50    value->types.i = atoi(literal);
51  }
52  else if (element->element_type == ELEMENT_REAL_LITERAL)
53  {
54    char* literal = element->element_value;
55
56    value->value_type = RAM_TYPE_REAL;
57    value->types.d = atof(literal);
58  }
59  else if (element->element_type == ELEMENT_TRUE || element->element_type == ELEMENT_FALSE)
60  {
61    char* literal = element->element_value;
62    value->value_type = RAM_TYPE_BOOLEAN;
63    if (strcmp(literal, "True") == 0)
64    {
65      value->types.i = 1;
66    }
67    else
68    {
69      assert(strcmp(literal, "False") == 0);
70      value->types.i = 0;
71    }
72
73  }
74  else if (element->element_type == ELEMENT_STR_LITERAL)
75  {
76    char* literal = element->element_value;
77
78    value->value_type = RAM_TYPE_STR;
79    value->types.s = literal;
80  }
81  else {
82    //
83    // identifier => variable
84    //
85    // old code:
86    assert(element->element_type == ELEMENT_IDENTIFIER);
87
88    char* var_name = element->element_value;
89
90    struct RAM_VALUE* ram_value = ram_read_cell_by_id(memory, var_name);
91
92    if (ram_value == NULL) {
93      printf("**SEMANTIC ERROR: name '%s' is not defined (line %d)\n", var_name, stmt->line);
94      return false;
```

```
95      }
96
97      value->value_type = ram_value->value_type;
98      if (value->value_type == RAM_TYPE_INT || value->value_type == RAM_TYPE_BOOLEAN)
99      {
100       value->types.i = ram_value->types.i;
101     }
102     else if (value->value_type == RAM_TYPE_REAL)
103     {
104       value->types.d = ram_value->types.d;
105     }
106     else if (value->value_type == RAM_TYPE_STR)
107     {
108       value->types.s = ram_value->types.s;
109     }
110   }
111
112   return true;
113 }
114
115
116 //
117 // get_unary_value
118 //
119 // Given a unary expr, returns the value that it represents.
120 // This could be the result of a literal 123 or the value
121 // from memory for an identifier such as "x". Unary values
122 // may have unary operators, such as + or -, applied.
123 // This value is "returned" via the reference parameter.
124 // Returns true if successful, false if not.
125 //
126 // Why would it fail? If the identifier does not exist in
127 // memory. This is a semantic error, and an error message is
128 // output before returning.
129 //
130 static bool get_unary_value(struct STMT* stmt, struct RAM* memory, struct UNARY_EXPR* unary, struct
    RAM_VALUE* value)
131 {
132   //
133   // we only have simple elements so far (no unary operators):
134   //
135   assert(unary->expr_type == UNARY_ELEMENT);
136
137   struct ELEMENT* element = unary->element;
138
139   bool success = get_element_value(stmt, memory, element, value);
140
141   return success;
142 }
```

```
143
144 //
145 // operator_plus
146 //
147 // Helper function to handle addition between lhs and rhs
148 //
149 static bool operator_plus(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
150 {
151   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
152   {
153     lhs->types.i = lhs->types.i + rhs.types.i;
154   }
155   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
156   {
157     lhs->types.d = lhs->types.d + rhs.types.d;
158   }
159   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
160   {
161     lhs->value_type = RAM_TYPE_REAL;
162     double lhs_real_val = lhs->types.i;
163     lhs->types.d = lhs_real_val + rhs.types.d;
164   }
165   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
166   {
167     lhs->types.d = lhs->types.d + rhs.types.i;
168   }
169   else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
170   {
171     char* concat = malloc(sizeof(char) * (strlen(lhs->types.s) + strlen(rhs.types.s) + 1));
172     strcpy(concat, lhs->types.s);
173     strcat(concat, rhs.types.s);
174
175     lhs->types.s = concat;
176   }
177   else
178   {
179     return false;
180   }
181   return true;
182 }
183
184 //
185 // operator_minus
186 //
187 // Helper function to handle subtraction between lhs and rhs
188 //
189 static bool operator_minus(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
190 {
191   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
```

```
192   {
193     lhs->types.i = lhs->types.i - rhs.types.i;
194   }
195   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
196   {
197     lhs->types.d = lhs->types.d - rhs.types.d;
198   }
199   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
200   {
201     lhs->value_type = RAM_TYPE_REAL;
202     double lhs_real_val = lhs->types.i;
203     lhs->types.d = lhs_real_val - rhs.types.d;
204   }
205   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
206   {
207     lhs->types.d = lhs->types.d - rhs.types.i;
208   }
209   else
210   {
211     return false;
212   }
213   return true;
214 }
215 
216 //
217 // operator_asterisk
218 //
219 // Helper function to handle multiplcation between lhs and rhs
220 //
221 static bool operator_asterisk(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
222 {
223   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
224   {
225     lhs->types.i = lhs->types.i * rhs.types.i;
226   }
227   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
228   {
229     lhs->types.d = lhs->types.d * rhs.types.d;
230   }
231   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
232   {
233     lhs->value_type = RAM_TYPE_REAL;
234     double lhs_real_val = lhs->types.i;
235     lhs->types.d = lhs_real_val * rhs.types.d;
236   }
237   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
238   {
239     lhs->types.d = lhs->types.d * rhs.types.i;
240   }
```

```
241   else
242   {
243     return false;
244   }
245   return true;
246 }
247 
248 //
249 // operator_power
250 //
251 // Helper function to handle power operation between lhs and rhs
252 //
253 static bool operator_power(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
254 {
255   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
256   {
257     lhs->types.i = pow(lhs->types.i, rhs.types.i);
258   }
259   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
260   {
261     lhs->types.d = pow(lhs->types.d, rhs.types.d);
262   }
263   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
264   {
265     lhs->value_type = RAM_TYPE_REAL;
266     double lhs_real_val = lhs->types.i;
267     lhs->types.d = pow(lhs_real_val, rhs.types.d);
268   }
269   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
270   {
271     lhs->types.d = pow(lhs->types.d, rhs.types.i);
272   }
273   else
274   {
275     return false;
276   }
277   return true;
278 }
279 
280 //
281 // operator_mod
282 //
283 // Helper function to handle modulus operation between lhs and rhs
284 //
285 static bool operator_mod(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
286 {
287   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
288   {
289     lhs->types.i = lhs->types.i % rhs.types.i;
```

```
290   }
291   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
292   {
293     lhs->types.d = fmod(lhs->types.d, rhs.types.d);
294   }
295   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
296   {
297     lhs->value_type = RAM_TYPE_REAL;
298     double lhs_real_val = lhs->types.i;
299     lhs->types.d = fmod(lhs_real_val, rhs.types.d);
300   }
301   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
302   {
303     lhs->types.d = fmod(lhs->types.d, rhs.types.i);
304   }
305   else
306   {
307     return false;
308   }
309   return true;
310 }
311
312 //
313 // operator_div
314 //
315 // Helper function to handle division between lhs and rhs
316 //
317 static bool operator_div(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
318 {
319   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
320   {
321     lhs->types.i = lhs->types.i / rhs.types.i;
322   }
323   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
324   {
325     lhs->types.d = lhs->types.d / rhs.types.d;
326   }
327   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
328   {
329     lhs->value_type = RAM_TYPE_REAL;
330     double lhs_real_val = lhs->types.i;
331     lhs->types.d = lhs_real_val / rhs.types.d;
332   }
333   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
334   {
335     lhs->types.d = lhs->types.d / rhs.types.i;
336   }
337   else
338   {
```

```
339     return false;
340   }
341   return true;
342 }
343
344 //
345 // operator_equal
346 //
347 // Helper function to equality operator between lhs and rhs
348 //
349 static bool operator_equal(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
350 {
351   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
352   {
353     lhs->value_type = RAM_TYPE_BOOLEAN;
354     if (lhs->types.i == rhs.types.i)
355     {
356       lhs->types.i = 1;
357     }
358     else
359     {
360       lhs->types.i = 0;
361     }
362   }
363   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
364   {
365     lhs->value_type = RAM_TYPE_BOOLEAN;
366     if (lhs->types.d == rhs.types.d)
367     {
368       lhs->types.i = 1;
369     }
370     else
371     {
372       lhs->types.i = 0;
373     }
374   }
375   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
376   {
377     lhs->value_type = RAM_TYPE_BOOLEAN;
378     if (lhs->types.i == rhs.types.d)
379     {
380       lhs->types.i = 1;
381     }
382     else
383     {
384       lhs->types.i = 0;
385     }
386   }
387   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
```

```
388   {
389     lhs->value_type = RAM_TYPE_BOOLEAN;
390     if (lhs->types.d == rhs.types.i)
391     {
392       lhs->types.i = 1;
393     }
394     else
395     {
396       lhs->types.i = 0;
397     }
398   }
399   else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
400   {
401     lhs->value_type = RAM_TYPE_BOOLEAN;
402     if (strcmp(lhs->types.s, rhs.types.s) == 0)
403     {
404       lhs->types.i = 1;
405     }
406     else
407     {
408       lhs->types.i = 0;
409     }
410   }
411   else
412   {
413     return false;
414   }
415   return true;
416 }
417 
418 //
419 // operator_not_equal
420 //
421 // Helper function to inequality operator between lhs and rhs
422 //
423 static bool operator_not_equal(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
424 {
425   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
426   {
427     lhs->value_type = RAM_TYPE_BOOLEAN;
428     if (lhs->types.i != rhs.types.i)
429     {
430       lhs->types.i = 1;
431     }
432     else
433     {
434       lhs->types.i = 0;
435     }
436   }
```

```
437   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
438   {
439     lhs->value_type = RAM_TYPE_BOOLEAN;
440     if (lhs->types.d != rhs.types.d)
441     {
442       lhs->types.i = 1;
443     }
444     else
445     {
446       lhs->types.i = 0;
447     }
448   }
449   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
450   {
451     lhs->value_type = RAM_TYPE_BOOLEAN;
452     if (lhs->types.i != rhs.types.d)
453     {
454       lhs->types.i = 1;
455     }
456     else
457     {
458       lhs->types.i = 0;
459     }
460   }
461   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
462   {
463     lhs->value_type = RAM_TYPE_BOOLEAN;
464     if (lhs->types.d != rhs.types.i)
465     {
466       lhs->types.i = 1;
467     }
468     else
469     {
470       lhs->types.i = 0;
471     }
472   }
473   else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
474   {
475     lhs->value_type = RAM_TYPE_BOOLEAN;
476     if (strcmp(lhs->types.s, rhs.types.s) != 0)
477     {
478       lhs->types.i = 1;
479     }
480     else
481     {
482       lhs->types.i = 0;
483     }
484   }
485   else
```

```
  {
    return false;
  }
  return true;
}

//
// operator_lt
//
// Helper function to "less than" operator between lhs and rhs
//
static bool operator_lt(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
{
  if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.i < rhs.types.i)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.d < rhs.types.d)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.i < rhs.types.d)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
```

```
  else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.d < rhs.types.i)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (strcmp(lhs->types.s, rhs.types.s) < 0)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else
  {
    return false;
  }
  return true;
}

//
// operator_lte
//
// Helper function to "less than or equal to" operator between lhs and rhs
//
static bool operator_lte(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
{
  if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.i <= rhs.types.i)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
```

```
584   }
585   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
586   {
587     lhs->value_type = RAM_TYPE_BOOLEAN;
588     if (lhs->types.d <= rhs.types.d)
589     {
590       lhs->types.i = 1;
591     }
592     else
593     {
594       lhs->types.i = 0;
595     }
596   }
597   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
598   {
599     lhs->value_type = RAM_TYPE_BOOLEAN;
600     if (lhs->types.i <= rhs.types.d)
601     {
602       lhs->types.i = 1;
603     }
604     else
605     {
606       lhs->types.i = 0;
607     }
608   }
609   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
610   {
611     lhs->value_type = RAM_TYPE_BOOLEAN;
612     if (lhs->types.d <= rhs.types.i)
613     {
614       lhs->types.i = 1;
615     }
616     else
617     {
618       lhs->types.i = 0;
619     }
620   }
621   else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
622   {
623     lhs->value_type = RAM_TYPE_BOOLEAN;
624     if (strcmp(lhs->types.s, rhs.types.s) <= 0)
625     {
626       lhs->types.i = 1;
627     }
628     else
629     {
630       lhs->types.i = 0;
631     }
632   }
```

```
633   else
634   {
635     return false;
636   }
637   return true;
638 }
639 
640 //
641 // operator_gt
642 //
643 // Helper function to "greater than" operator between lhs and rhs
644 //
645 static bool operator_gt(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
646 {
647   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
648   {
649     lhs->value_type = RAM_TYPE_BOOLEAN;
650     if (lhs->types.i > rhs.types.i)
651     {
652       lhs->types.i = 1;
653     }
654     else
655     {
656       lhs->types.i = 0;
657     }
658   }
659   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
660   {
661     lhs->value_type = RAM_TYPE_BOOLEAN;
662     if (lhs->types.d > rhs.types.d)
663     {
664       lhs->types.i = 1;
665     }
666     else
667     {
668       lhs->types.i = 0;
669     }
670   }
671   else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
672   {
673     lhs->value_type = RAM_TYPE_BOOLEAN;
674     if (lhs->types.i > rhs.types.d)
675     {
676       lhs->types.i = 1;
677     }
678     else
679     {
680       lhs->types.i = 0;
681     }
```

```
682   }
683   else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
684   {
685     lhs->value_type = RAM_TYPE_BOOLEAN;
686     if (lhs->types.d > rhs.types.i)
687     {
688       lhs->types.i = 1;
689     }
690     else
691     {
692       lhs->types.i = 0;
693     }
694   }
695   else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
696   {
697     lhs->value_type = RAM_TYPE_BOOLEAN;
698     if (strcmp(lhs->types.s, rhs.types.s) > 0)
699     {
700       lhs->types.i = 1;
701     }
702     else
703     {
704       lhs->types.i = 0;
705     }
706   }
707   else
708   {
709     return false;
710   }
711   return true;
712 }
713
714 //
715 // operator_gte
716 //
717 // Helper function to "greater than or equal to" operator between lhs and rhs
718 //
719 static bool operator_gte(struct RAM_VALUE* lhs, struct RAM_VALUE rhs)
720 {
721   if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_INT)
722   {
723     lhs->value_type = RAM_TYPE_BOOLEAN;
724     if (lhs->types.i >= rhs.types.i)
725     {
726       lhs->types.i = 1;
727     }
728     else
729     {
730       lhs->types.i = 0;
```

```
    }
  }
  else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_REAL)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.d >= rhs.types.d)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_INT && rhs.value_type == RAM_TYPE_REAL)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.i >= rhs.types.d)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_REAL && rhs.value_type == RAM_TYPE_INT)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (lhs->types.d >= rhs.types.i)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
  }
  else if (lhs->value_type == RAM_TYPE_STR && rhs.value_type == RAM_TYPE_STR)
  {
    lhs->value_type = RAM_TYPE_BOOLEAN;
    if (strcmp(lhs->types.s, rhs.types.s) >= 0)
    {
      lhs->types.i = 1;
    }
    else
    {
      lhs->types.i = 0;
    }
```

```
780   }
781   else
782   {
783     return false;
784   }
785   return true;
786 }
787 
788 //
789 // execute_binary_expr
790 //
791 // Given two values and an operator, performs the operation
792 // and updates the value in the lhs (which can be updated
793 // because a pointer to the value is passed in). Returns
794 // true if successful and false if not.
795 //
796 static bool execute_binary_expr(struct RAM_VALUE* lhs, int operator, struct RAM_VALUE rhs)
797 {
798   assert(operator != OPERATOR_NO_OP);
799 
800   //
801   // perform the operation:
802   //
803   switch (operator)
804   {
805   case OPERATOR_PLUS:
806     if (!(operator_plus(lhs, rhs)))
807     {
808       return false;
809     }
810     break;
811 
812   case OPERATOR_MINUS:
813     if (!(operator_minus(lhs, rhs)))
814     {
815       return false;
816     }
817     break;
818 
819   case OPERATOR_ASTERISK:
820     if (!(operator_asterisk(lhs, rhs)))
821     {
822       return false;
823     }
824     break;
825 
826   case OPERATOR_POWER:
827     if (!(operator_power(lhs, rhs)))
828     {
```

```
829       return false;
830     }
831     break;
832
833   case OPERATOR_MOD:
834     if (!(operator_mod(lhs, rhs)))
835     {
836       return false;
837     }
838
839     break;
840
841   case OPERATOR_DIV:
842     if (!(operator_div(lhs, rhs)))
843     {
844       return false;
845     }
846     break;
847
848   case OPERATOR_EQUAL:
849     if (!(operator_equal(lhs, rhs)))
850     {
851       return false;
852     }
853     break;
854
855   case OPERATOR_NOT_EQUAL:
856     if (!(operator_not_equal(lhs, rhs)))
857     {
858       return false;
859     }
860     break;
861
862   case OPERATOR_LT:
863     if (!(operator_lt(lhs, rhs)))
864     {
865       return false;
866     }
867     break;
868
869   case OPERATOR_LTE:
870     if (!(operator_lte(lhs, rhs)))
871     {
872       return false;
873     }
874     break;
875
876   case OPERATOR_GT:
877     if (!(operator_gt(lhs, rhs)))
```

```
878     {
879       return false;
880     }
881     break;
882
883   case OPERATOR_GTE:
884     if (!(operator_gte(lhs, rhs)))
885     {
886       return false;
887     }
888     break;
889
890   default:
891     //
892     // did we miss something?
893     //
894     printf("**EXECUTION ERROR: unexpected operator (%d) in execute_binary_expr\n", operator);
895     return false;
896   }
897
898   return true;
899 }
900
901 //
902 // is_zeros
903 //
904 // Checks if a string is all zeros
905 //
906 static bool is_zeros(char* num)
907 {
908   for (int i = 0; i < strlen(num); i++)
909   {
910     if (num[i] != '0' && num[i] != '.')
911     {
912       return false;
913     }
914   }
915   return true;
916 }
917
918 //
919 // compute_value
920 //
921 // Stores the result of an expression in the value passed by address, returning success
922 // boolean
923 //
924 static bool compute_value(struct STMT* stmt, struct RAM* memory, struct RAM_VALUE* value)
925 {
926   struct VALUE_EXPR* expr = malloc(sizeof(struct VALUE_EXPR));
```

```
927   if (expr == NULL)
928   {
929     return false;
930   }
931   if (stmt->stmt_type == STMT_WHILE_LOOP)
932   {
933     struct STMT_WHILE_LOOP* while_loop = stmt->types.while_loop;
934     expr = while_loop->condition;
935   }
936   else
937   {
938     assert(stmt->stmt_type == STMT_ASSIGNMENT);
939
940     struct STMT_ASSIGNMENT* assign = stmt->types.assignment;
941     expr = assign->rhs->types.expr;
942   }
943   //
944   // we always have a LHS:
945   //
946   assert(expr->lhs != NULL);
947
948   bool success = get_unary_value(stmt, memory, expr->lhs, value);
949
950   if (!success)  // semantic error? If so, return now:
951     return false;
952
953   //
954   // do we have a binary expression?
955   //
956   if (expr->isBinaryExpr)
957   {
958     assert(expr->rhs != NULL);  // we must have a RHS
959     assert(expr->operator != OPERATOR_NO_OP);  // we must have an operator
960
961     struct RAM_VALUE rhs_value;
962
963     success = get_unary_value(stmt, memory, expr->rhs, &rhs_value);
964
965     if (!success) {  // semantic error? If so, return now:
966       return false;
967     }
968
969     //
970     // perform the operation, updating value:
971     //
972     bool success = execute_binary_expr(value, expr->operator, rhs_value);
973
974     if (!success) {
975       printf("**SEMANTIC ERROR: invalid operand types (line %i)\n", stmt->line);
```

```
976       return false;
977     }
978   }
979
980   return true;
981 }
982
983
984
985 //
986 // execute_assignment
987 //
988 // Executes an assignment statement, returning true if
989 // successful and false if not (an error message will be
990 // output before false is returned, so the caller doesn't
991 // need to output anything).
992 //
993 // Examples: x = 123
994 //           y = x ** 2
995 //
996 static bool execute_assignment(struct STMT* stmt, struct RAM* memory)
997 {
998   struct STMT_ASSIGNMENT* assign = stmt->types.assignment;
999
1000  char* var_name = assign->var_name;
1001
1002  //
1003  // no pointers yet:
1004  //
1005  assert(assign->isPtrDeref == false);
1006
1007  if (assign->rhs->value_type == VALUE_FUNCTION_CALL)
1008  {
1009    struct VALUE_FUNCTION_CALL* func_call = assign->rhs->types.function_call;
1010    if (strcmp(func_call->function_name, "input") == 0)
1011    {
1012      if (func_call->parameter != NULL)
1013        {
1014          // assumed that element_value is always a string literal
1015          char* element_value = func_call->parameter->element_value;
1016
1017          printf("%s", element_value);
1018        }
1019
1020        char line[256];
1021        fgets(line, sizeof(line), stdin);
1022        // delete EOL chars from input:
1023        line[strcspn(line, "\r\n")] = '\0';
1024
```

```
1025        struct RAM_VALUE value;
1026        value.value_type = RAM_TYPE_STR;
1027        char* inp_val = malloc(sizeof(char) * (strlen(line) + 1));
1028        strcpy(inp_val, line);
1029        value.types.s = inp_val;
1030
1031        bool success = ram_write_cell_by_id(memory, value, var_name);
1032        return success;
1033      }
1034      else if (strcmp(func_call->function_name, "int") == 0)
1035      {
1036        struct RAM_VALUE value;
1037        value.value_type = RAM_TYPE_INT;
1038        char* stored_str_value = ram_read_cell_by_id(memory, func_call->parameter->element_value)-
      >types.s;
1039        int conv_int_value = atoi(stored_str_value);
1040        if (conv_int_value == 0 && !is_zeros(stored_str_value))
1041        {
1042          printf("**SEMANTIC ERROR: invalid string for int() (line %i)\n", stmt->line);
1043          return false;
1044        }
1045        else
1046        {
1047          value.types.i = conv_int_value;
1048          bool success = ram_write_cell_by_id(memory, value, var_name);
1049          return success;
1050        }
1051      }
1052      else
1053      {
1054        assert(strcmp(func_call->function_name, "float") == 0);
1055
1056        struct RAM_VALUE value;
1057        value.value_type = RAM_TYPE_REAL;
1058        char* stored_str_value = ram_read_cell_by_id(memory, func_call->parameter->element_value)-
      >types.s;
1059        double conv_float_value = atof(stored_str_value);
1060        if (conv_float_value == 0 && !is_zeros(stored_str_value))
1061        {
1062          printf("**SEMANTIC ERROR: invalid string for float() (line %i)\n", stmt->line);
1063          return false;
1064        }
1065        else
1066        {
1067          value.types.d = conv_float_value;
1068          bool success = ram_write_cell_by_id(memory, value, var_name);
1069          return success;
1070        }
1071      }
```

```
1072    }
1073    else
1074    {
1075      assert(assign->rhs->value_type == VALUE_EXPR);
1076
1077      struct RAM_VALUE value;
1078
1079      bool success = compute_value(stmt, memory, &value);
1080      if (!success)
1081      {
1082        return false;
1083      }
1084
1085      //
1086      // write the value to memory:
1087      //
1088      success = ram_write_cell_by_id(memory, value, var_name);
1089
1090      return success;
1091    }
1092
1093  }
1094
1095  //
1096  // execute_print
1097  //
1098  // Helper function for execute_function_call which handles printing
1099  //
1100  static bool execute_print(struct STMT* stmt, struct STMT_FUNCTION_CALL* call, struct RAM* memory)
1101  {
1102    char* function_name = call->function_name;
1103    assert(strcmp(function_name, "print") == 0);
1104
1105    if (call->parameter == NULL) {
1106      printf("\n");
1107    }
1108    else {
1109      //
1110      // we have a parameter, which type of parameter?
1111      // Note that a parameter is a simple element, i.e.
1112      // identifier or literal (or True, False, None):
1113      //
1114      char* element_value = call->parameter->element_value;
1115
1116      if (call->parameter->element_type == ELEMENT_STR_LITERAL) {
1117        printf("%s\n", element_value);
1118      }
1119      else {
1120        struct RAM_VALUE value;
```

```
1121
1122
1123      bool success = get_element_value(stmt, memory, call->parameter, &value);
1124
1125      if (!success)
1126        return false;
1127
1128      if (value.value_type == RAM_TYPE_INT)
1129      {
1130        printf("%d\n", value.types.i);
1131      }
1132      else if (value.value_type == RAM_TYPE_REAL)
1133      {
1134        printf("%f\n", value.types.d);
1135      }
1136      else if (value.value_type == RAM_TYPE_STR)
1137      {
1138        printf("%s\n", value.types.s);
1139      }
1140      else if (value.value_type == RAM_TYPE_BOOLEAN)
1141      {
1142        if (value.types.i == 0)
1143        {
1144          printf("False\n");
1145        }
1146        else
1147        {
1148          printf("True\n");
1149        }
1150      }
1151    }
1152  }//else
1153
1154  return true;
1155 }
1156
1157 //
1158 // execute_function_call
1159 //
1160 // Executes a function call statement, returning true if
1161 // successful and false if not (an error message will be
1162 // output before false is returned, so the caller doesn't
1163 // need to output anything).
1164 //
1165 // Examples: print()
1166 //           print(x)
1167 //           print(123)
1168 //
1169 static bool execute_function_call(struct STMT* stmt, struct RAM* memory)
```

```
1170 {
1171   struct STMT_FUNCTION_CALL* call = stmt->types.function_call;
1172
1173   //
1174   // for now we are assuming it's a call to print:
1175   //
1176   char* function_name = call->function_name;
1177
1178   if (strcmp(function_name, "print") == 0)
1179   {
1180     return execute_print(stmt, call, memory);
1181   }
1182   else
1183   {
1184     return false;
1185   }
1186 }
1187
1188
1189 //
1190 // Public functions:
1191 //
1192
1193 //
1194 // execute
1195 //
1196 // Given a nuPython program graph and a memory,
1197 // executes the statements in the program graph.
1198 // If a semantic error occurs (e.g. type error),
1199 // an error message is output, execution stops,
1200 // and the function returns.
1201 //
1202 void execute(struct STMT* program, struct RAM* memory)
1203 {
1204   struct STMT* stmt = program;
1205
1206   //
1207   // traverse through the program statements:
1208   //
1209   while (stmt != NULL) {
1210
1211     if (stmt->stmt_type == STMT_ASSIGNMENT) {
1212
1213       bool success = execute_assignment(stmt, memory);
1214
1215       if (!success)
1216         return;
1217
1218       stmt = stmt->types.assignment->next_stmt;  // advance
```

```
1219     }
1220     else if (stmt->stmt_type == STMT_FUNCTION_CALL) {
1221
1222       bool success = execute_function_call(stmt, memory);
1223
1224       if (!success)
1225         return;
1226
1227       stmt = stmt->types.function_call->next_stmt;
1228     }
1229     else if (stmt->stmt_type == STMT_WHILE_LOOP)
1230     {
1231       struct RAM_VALUE value;
1232       value.value_type = RAM_TYPE_BOOLEAN;
1233       bool success = compute_value(stmt, memory, &value);
1234       if (!success)
1235       {
1236         return;
1237       }
1238
1239       if (value.value_type == RAM_TYPE_BOOLEAN && value.types.i == 1)
1240       {
1241         stmt = stmt->types.while_loop->loop_body;
1242       }
1243       else
1244       {
1245         stmt = stmt->types.while_loop->next_stmt;
1246       }
1247     }
1248     else {
1249       assert(stmt->stmt_type == STMT_PASS);
1250
1251       //
1252       // nothing to do!
1253       //
1254
1255       stmt = stmt->types.pass->next_stmt;
1256     }
1257   }//while
1258
1259   //
1260   // done:
1261   //
1262   return;
1263 }
1264
```