Project 02 - execute

● Graded

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

Question 2

Manual review **20** / 20 pts

- ✔ **+ 2 pts** "execute.c" header comment has description
- ✔ **+ 2 pts** "execute.c" header comment has name
- ✔ **+ 1 pt** "execute.c" header comment school, course, etc.
- ✔ **+ 4 pts** Definition and use of execute_function_call( )
- ✔ **+ 1 pt** has header comment
- ✔ **+ 4 pts** Definition and use of execute_assignment( )
- ✔ **+ 1 pt** has header comment
- ✔ **+ 4 pts** Definition and use of additional helper function in support of binary expressions (non-trivial with at least 10 lines of meaningful code)
- ✔ **+ 1 pt** has header comment

Autograder Results

Autograder Output

```
**********************************************************
This is submission #2
Submitted @ 23:4 on 2024-1-19 (Chicago time)

Submission history:
 Submission #1: score=70, submitted @ 22:5 on 2024-1-19 (Chicago time)

Total # of valid submissions so far: 1
# of valid submissions since midnight: 1
# of minutes since last valid submission: 59
**********************************************************
You have 1 submission this 24-hr period.


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

This is submission #2 in current time period

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

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


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


** Test Input:
#
# test03.py
#
# a nuPython program of binary expressions
#
print("")
print("TEST CASE: test03.py")
print("")

x = 3 * 4     # 12
y = x ** 2    # 144
z = 288 / y   # 2
x = 5         # overwrite x to now be 5
remainder = x % z    # 1

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

```
print(remainder)

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


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('')
7: print('TEST CASE: test03.py')
8: print('')
9:
10: x = 3 * 4
11: y = x ** 2
12: z = 288 / y
13: x = 5
14: remainder = x % z
15:
16: print(x)
17: print(y)
18: print(z)
19: print(remainder)
20:
21: print('')
22: print('DONE')
23: print('')
24: $
**END PRINT**
**executing...

TEST CASE: test03.py

5
144
2
1

DONE

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

```
Capacity: 4
Num values: 4
Contents:
 0: x, int, 5
 1: y, int, 144
 2: z, int, 2
 3: remainder, int, 1
**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:
#
# test01.py
#
# a simple nuPython program of print("...") calls
#
print("")
print("TEST CASE: test01.py")
print("")

print('a simple program')
print('that')
print("consists")
print('of')
print('calls to print(STRING)')

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

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('')
7: print('TEST CASE: test01.py')
8: print('')
9:
10: print('a simple program')
11: print('that')
12: print('consists')
13: print('of')
14: print('calls to print(STRING)')
15:
16: print('')
17: print('DONE')
18: print('')
19: $
**END PRINT**
**executing...

TEST CASE: test01.py

a simple program
that
consists
of
calls to print(STRING)

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:
#
# test02.py
#
# a nuPython program of simple assignment and print(variable)
#
print()
print("TEST CASE: test02.py")
print()

x = 123
y = 456
print(x)
print(y)

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


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print()
7: print('TEST CASE: test02.py')
8: print()
9:
10: x = 123
```

```
11: y = 456
12: print(x)
13: print(y)
14:
15: print()
16: print('DONE')
17: print()
18: $
**END PRINT**
**executing...

TEST CASE: test02.py

123
456

DONE

**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
Contents:
 0: x, int, 123
 1: y, int, 456
**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:
#
```

```
# test03.py
#
# a nuPython program of binary expressions
#
print("")
print("TEST CASE: test03.py")
print("")

x = 3 * 4     # 12
y = x ** 2    # 144
z = 288 / y   # 2
x = 5         # overwrite x to now be 5
remainder = x % z    # 1

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

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


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('')
7: print('TEST CASE: test03.py')
8: print('')
9:
10: x = 3 * 4
11: y = x ** 2
12: z = 288 / y
13: x = 5
14: remainder = x % z
15:
16: print(x)
17: print(y)
18: print(z)
19: print(remainder)
20:
21: print('')
```

```
22: print('DONE')
23: print('')
24: $
**END PRINT**
**executing...

TEST CASE: test03.py

5
144
2
1

DONE

**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
 0: x, int, 5
 1: y, int, 144
 2: z, int, 2
 3: remainder, int, 1
**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:
#
# test04.py
```

```
#
# a nuPython program with semantic error
#
print("")
print("TEST CASE: test04.py")
print("")

x = 123
print(y)   # error
y = 456
print(y)

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

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('')
7: print('TEST CASE: test04.py')
8: print('')
9:
10: x = 123
11: print(y)
12: y = 456
13: print(y)
14:
15: print('')
16: print('DONE')
17: print('')
18: $
**END PRINT**
**executing...

TEST CASE: test04.py

**SEMANTIC ERROR: name 'y' is not defined (line 11)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 1
```

```
Contents:
 0: x, 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 4 **
************************************************************************


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


** Test Input:
#
# test05.py
#
# a nuPython program of binary expr with semantic error
#
print("")
print("TEST CASE: test05.py")
print("")

x = 3 * 4     # 12
y = x ** 2    # 144
z = 288 / fred   # ERROR
x = 5         # overwrite x to now be 5
remainder = x % z    # 1

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

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

```
** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('')
7: print('TEST CASE: test05.py')
8: print('')
9:
10: x = 3 * 4
11: y = x ** 2
12: z = 288 / fred
13: x = 5
14: remainder = x % z
15:
16: print(x)
17: print(y)
18: print(z)
19: print(remainder)
20:
21: print('')
22: print('DONE')
23: print('')
24: $
**END PRINT**
**executing...

TEST CASE: test05.py

**SEMANTIC ERROR: name 'fred' is not defined (line 12)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
Contents:
 0: x, int, 12
 1: y, int, 144
**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:
```
#
# test06.py
#
# a nuPython program of binary expr with semantic error
#
print("")
print("TEST CASE: test06.py")
print("")

x = 3 * 4     # 12
y = abc ** 2    # ERROR
z = 288 / y   # 2
x = 5         # overwrite x to now be 5
remainder = x % z    # 1

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

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

** Your output (first 600 lines) **
```
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
```

```
5:
6: print('')
7: print('TEST CASE: test06.py')
8: print('')
9:
10: x = 3 * 4
11: y = abc ** 2
12: z = 288 / y
13: x = 5
14: remainder = x % z
15:
16: print(x)
17: print(y)
18: print(z)
19: print(remainder)
20:
21: print('')
22: print('DONE')
23: print('')
24: $
**END PRINT**
**executing...

TEST CASE: test06.py

**SEMANTIC ERROR: name 'abc' is not defined (line 11)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 1
Contents:
 0: x, int, 12
**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:
#
# test07.py
#
# a nuPython program of binary expressions
#
print("TEST CASE: test07.py")
print()

# random comment

x = 100        # 100
y = x - 140    # -40

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

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)

# another random comment

print()
```

```
print("DONE")


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('TEST CASE: test07.py')
7: print()
8:
9:
10:
11: x = 100
12: y = x - 140
13:
14: print('x is:')
15: print(x)
16: print('y is:')
17: print(y)
18:
19: z = y * y
20: print('and now for z:')
21: print(z)
22:
23: test1 = x + y
24: x = 1
25: test2 = x + y
26: y = 123
27: test3 = x * y
28: test4 = z / 3
29: test5 = z % 11
30: x = 4
31: test6 = test5 ** x
32:
33: print('test variables:')
34: print(test1)
35: print(test2)
36: print(test3)
37: print(test4)
38: print(test5)
39: print(test6)
40:
41:
42:
```

```
43: print()
44: print('DONE')
45: $
**END PRINT**
**executing...
TEST CASE: test07.py

x is:
100
y is:
-40
and now for z:
1600
test variables:
60
-39
123
533
5
625

DONE
**done
**MEMORY PRINT**
Capacity: 16
Num values: 9
Contents:
 0: x, int, 4
 1: y, int, 123
 2: z, int, 1600
 3: test1, int, 60
 4: test2, int, -39
 5: test3, int, 123
 6: test4, int, 533
 7: test5, int, 5
 8: test6, int, 625
**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:
#
# test07.py
#
# a nuPython program of binary expressions
#
print("TEST CASE: test07.py")
print()

# random comment

x = 100        # 100
y = x - 140    # -40

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

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
test7 = x * xx       # ERROR

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

```
print(test6)
print(test7)

# another random comment

print()
print("DONE")


** Your output (first 600 lines) **
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1:
2:
3:
4:
5:
6: print('TEST CASE: test07.py')
7: print()
8:
9:
10:
11: x = 100
12: y = x - 140
13:
14: print('x is:')
15: print(x)
16: print('y is:')
17: print(y)
18:
19: z = y * y
20: print('and now for z:')
21: print(z)
22:
23: test1 = x + y
24: x = 1
25: test2 = x + y
26: y = 123
27: test3 = x * y
28: test4 = z / 3
29: test5 = z % 11
30: x = 4
31: test6 = test5 ** x
32: test7 = x * xx
33:
34: print('test variables:')
35: print(test1)
36: print(test2)
```

```
37: print(test3)
38: print(test4)
39: print(test5)
40: print(test6)
41: print(test7)
42:
43:
44:
45: print()
46: print('DONE')
47: $
**END PRINT**
**executing...
TEST CASE: test07.py

x is:
100
y is:
-40
and now for z:
1600
**SEMANTIC ERROR: name 'xx' is not defined (line 32)
**done
**MEMORY PRINT**
Capacity: 16
Num values: 9
Contents:
 0: x, int, 4
 1: y, int, 123
 2: z, int, 1600
 3: test1, int, 60
 4: test2, int, -39
 5: test3, int, 123
 6: test4, int, 533
 7: test5, int, 5
 8: test6, int, 625
**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 **
****

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 (int variables) -- yay, output correct!

Test 3: test03.py

Test 3: test03.py (binary expressions) -- yay, output correct!

Test 4: test04.py

Test 4: test04.py (semantic error) -- yay, output correct!

Test 5: test05.py

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

Test 6: test06.py

Test 6: test06.py (semantic error) -- yay, output correct!

Test 7: test07.py

Test 7: test07.py (lots of statements) -- yay, output correct!

Test 8: test08.py

Test 8: test08.py (semantic error) -- yay, output correct!

Submitted Files

execute.c Download

```c
/*execute.c*/

//
// << WHAT IS THE PURPOSE OF THIS FILE??? >>
// Executing nuPython programs
// << WHAT IS YOUR NAME >> Ishan Mukherjee
// << WHAT SCHOOL IS THIS >> Northwestern University
// << WHAT COURSE IS THIS >> CS 211
// << WHAT QUARTER IS THIS >> Winter 2024
//
// Starter code: Prof. Joe Hummel
//

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

#include "programgraph.h"
#include "ram.h"
#include "execute.h"

//
// semantic_error
//
// Prints a semantic error message given the name of the identifier and line number
//

static void semantic_error(char* identifier_name, int line)
{
  printf("**SEMANTIC ERROR: name '%s' is not defined (line %i)\n", identifier_name, line);
}


//
// execute_function_call
//
// Handles (only print for now) function calls, given statement and memory pointers
//

static bool execute_function_call(struct STMT* statement, struct RAM* memory)
{
  if (statement->types.function_call->parameter == NULL)
  {
```

```
47      printf("\n");
48      return true;
49    }
50    else if (statement->types.function_call->parameter->element_type == ELEMENT_IDENTIFIER)
51    {
52      if (ram_read_cell_by_id(memory, statement->types.function_call->parameter->element_value) !=
    NULL)
53      {
54        printf("%d\n", ram_read_cell_by_id(memory, statement->types.function_call->parameter-
    >element_value)->types.i);
55        return true;
56      }
57      else
58      {
59        semantic_error(statement->types.function_call->parameter->element_value, statement->line);
60        return false;
61      }
62    }
63    else if (statement->types.function_call->parameter->element_type == ELEMENT_INT_LITERAL)
64    {
65      printf("%d\n", atoi(statement->types.function_call->parameter->element_value));
66      return true;
67    }
68    else if (statement->types.function_call->parameter->element_type == ELEMENT_STR_LITERAL)
69    {
70      printf("%s\n", statement->types.function_call->parameter->element_value);
71      return true;
72    }
73    else
74    {
75      semantic_error(statement->types.function_call->parameter->element_value, statement->line);
76      return false;
77    }
78  }
79
80  //
81  // update_res_val
82  //
83  // Helper function for execute_assignment, which updates the result value given lhs and rhs values
84  // and operator, and returns false if invalid operation found or true if executed successfully
85  //
86
87  static bool update_res_val(struct RAM_VALUE* result_val, struct RAM_VALUE lhs_val, struct RAM_VALUE
    rhs_val, int op)
88  {
89    // print debugging code:
90    // printf("lhs: %i rhs: %i\n", lhs_val.types.i, rhs_val.types.i);
91
92    // if-else calculates result based on the operator
```

```
93    if (op == OPERATOR_PLUS)
94    {
95      result_val->types.i = lhs_val.types.i + rhs_val.types.i;
96    }
97    else if (op == OPERATOR_MINUS)
98    {
99      result_val->types.i = lhs_val.types.i - rhs_val.types.i;
100   }
101   else if (op == OPERATOR_ASTERICK)
102   {
103     result_val->types.i = lhs_val.types.i * rhs_val.types.i;
104   }
105   else if (op == OPERATOR_DIV)
106   {
107     result_val->types.i = lhs_val.types.i / rhs_val.types.i;
108   }
109   else if (op == OPERATOR_MOD)
110   {
111     result_val->types.i = lhs_val.types.i % rhs_val.types.i;
112   }
113   else if (op == OPERATOR_POWER)
114   {
115     result_val->types.i = (int) pow(lhs_val.types.i, rhs_val.types.i);
116   }
117   else
118   {
119     return false;
120   }
121   return true;
122 }
123
124 //
125 // execute_assignment
126 //
127 // Handles assignment of variables, given statement and memory pointers
128 //
129
130 static bool execute_assignment(struct STMT* statement, struct RAM* memory)
131 {
132   struct RAM_VALUE result_val;
133   result_val.value_type = RAM_TYPE_INT;
134   result_val.types.i = 0; // initializing
135
136   // finding lhs of operator
137   struct RAM_VALUE lhs_val;
138   lhs_val.value_type = RAM_TYPE_INT;
139   if (statement->types.assignment->rhs->types.expr->lhs->element->element_type ==
    ELEMENT_IDENTIFIER)
140   {
```

```
141     if (ram_read_cell_by_id(memory, statement->types.assignment->rhs->types.expr->lhs->element->element_value) != NULL)
142     {
143       lhs_val.types.i = ram_read_cell_by_id(memory, statement->types.assignment->rhs->types.expr->lhs->element->element_value)->types.i;
144     }
145     else
146     {
147       semantic_error(statement->types.assignment->rhs->types.expr->lhs->element->element_value, statement->line);
148       return false;
149     }
150   }
151   else if (statement->types.assignment->rhs->types.expr->lhs->element->element_type == ELEMENT_INT_LITERAL)
152   {
153     lhs_val.types.i = atoi(statement->types.assignment->rhs->types.expr->lhs->element->element_value);
154   }
155   else
156   {
157     semantic_error(statement->types.assignment->rhs->types.expr->lhs->element->element_value, statement->line);
158     return false;
159   }
160
161   // finding rhs of operator and result value
162   struct RAM_VALUE rhs_val;
163   // if a binary expression
164   if (statement->types.assignment->rhs->types.expr->isBinaryExpr)
165   {
166     rhs_val.value_type = RAM_TYPE_INT;
167     rhs_val.types.i = 0; // initializing
168     // if rhs of operator is identifier
169     if (statement->types.assignment->rhs->types.expr->rhs->element->element_type == ELEMENT_IDENTIFIER)
170     {
171       if (ram_read_cell_by_id(memory, statement->types.assignment->rhs->types.expr->rhs->element->element_value) != NULL)
172       {
173         rhs_val.types.i = ram_read_cell_by_id(memory, statement->types.assignment->rhs->types.expr->rhs->element->element_value)->types.i;
174       }
175       else
176       {
177         semantic_error(statement->types.assignment->rhs->types.expr->rhs->element->element_value, statement->line);
178         return false;
179       }
180     }
```

```
181     // if rhs of operator is int literal
182     else if (statement->types.assignment->rhs->types.expr->rhs->element->element_type ==
    ELEMENT_INT_LITERAL)
183     {
184       rhs_val.types.i = atoi(statement->types.assignment->rhs->types.expr->rhs->element-
    >element_value);
185     }
186     // if rhs of operator is a secret third thing
187     else
188     {
189       return false;
190     }
191
192     // finding result val
193     //
194     // if updating the result val ended in disaster (aka returned false)
195     if (!update_res_val(&result_val, lhs_val, rhs_val, statement->types.assignment->rhs->types.expr-
    >operator))
196     {
197       // abort abort
198       return false;
199     }
200   }
201   // if not a binary expression
202   else
203   {
204     result_val = lhs_val;
205   }
206
207   ram_write_cell_by_id(memory, result_val, statement->types.assignment->var_name);
208
209   return true;
210 }
211
212
213 //
214 // Public functions:
215 //
216
217 //
218 // execute
219 //
220 // Given a nuPython program graph and a memory,
221 // executes the statements in the program graph.
222 // If a semantic error occurs (e.g. type error),
223 // and error message is output, execution stops,
224 // and the function returns.
225 //
226 void execute(struct STMT* program, struct RAM* memory)
```

```
{
  struct STMT* stmt = program;
  while (stmt != NULL)
  {
    // assignment
    if (stmt->stmt_type == STMT_ASSIGNMENT)
    {
      int line = stmt->line;
      char* var_name = stmt->types.assignment->var_name;
      if (!execute_assignment(stmt, memory))
      {
        return;
      }
      stmt = stmt->types.assignment->next_stmt; // advance to next statement
    }
    // function call
    else if (stmt->stmt_type == STMT_FUNCTION_CALL)
    {
      if (!execute_function_call(stmt, memory))
      {
        return;
      }
      stmt = stmt->types.function_call->next_stmt;
    }
    // pass
    else
    {
      assert(stmt->stmt_type == STMT_PASS);
      stmt = stmt->types.pass->next_stmt;
    }
  }
}

```

main.c

Download

```c
/*main.c*/

//
// Project 01: main program to test scanner for nuPython
//
// Prof. Joe Hummel
// Northwestern University
// CS 211
//

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

#include "token.h"    // token defs
#include "scanner.h"  // scanner

#include "parser.h"
#include "programgraph.h"
#include "ram.h"
#include "execute.h"


//
// main
//
// usage: program.exe [filename.py]
//
// Student: Ishan Mukherjee
//
// If a filename is given, the file is opened and serves as
// input to the program execution function. 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) {
    //
    // no args, just the program name:
    //
    input = stdin;
    keyboardInput = true;
```

```
47    }
48    else {
49      //
50      // assume 2nd arg is a nuPython file:
51      //
52      char* filename = argv[1];
53
54      input = fopen(filename, "r");
55
56      if (input == NULL) // unable to open:
57      {
58        printf("**ERROR: unable to open input file '%s' for input.\n", filename);
59        return 0;
60      }
61
62      keyboardInput = false;
63    }
64
65    //
66    // input the tokens, either from keyboard or the given nuPython
67    // file; the "input" variable controls the source. the scanner will
68    // stop and return EOS when the user enters $ or we reach EOF on
69    // the nuPython file:
70    //
71    // int lineNumber = -1;
72    // int colNumber = -1;
73    // char value[256] = "";
74    // struct Token T;
75
76    //
77    // setup to start scanning:
78    //
79    // scanner_init(&lineNumber, &colNumber, value);
80
81    if (keyboardInput)  // prompt the user if appropriate:
82    {
83      printf("nuPython input (enter $ when you're done)>\n");
84    }
85
86    //
87    // call parser to check program syntax:
88    //
89    parser_init();
90    struct TokenQueue* tokens = parser_parse(input);
91    if (tokens == NULL)
92    {
93      //
94      // program has a syntax error, error msg already output:
95      //
```

```
96    }
97    else
98    {
99      //
100     // parsing successful, now build program graph:
101     //
102     printf("**no syntax errors...\n");
103     printf("**building program graph...\n");
104     struct STMT* program = programgraph_build(tokens);
105     programgraph_print(program);
106     printf("**executing...\n");
107     struct RAM* memory = ram_init();
108     execute(program, memory);
109     printf("**done\n");
110     ram_print(memory);
111   }
112   //
113   // call scanner to process input token by token until we see ; or $
114   //
115   // T = scanner_nextToken(input, &lineNumber, &colNumber, value);
116
117   // while (T.id != nuPy_EOS)
118   // {
119   //   printf("Token %d ('%s') @ (%d, %d)\n", T.id, value, T.line, T.col);
120
121   //   T = scanner_nextToken(input, &lineNumber, &colNumber, value);
122   // }
123
124   // // output that last token:
125   // printf("Token %d ('%s') @ (%d, %d)\n", T.id, value, T.line, T.col);
126
127   //
128   // done:
129   //
130   if (!keyboardInput)
131     fclose(input);
132
133   return 0;
134 }
135
```