

Project #02 (v1.2)

Assignment: nuPython program execution (part 01)

Submission: Gradescope

Policy: individual work only, late work is accepted

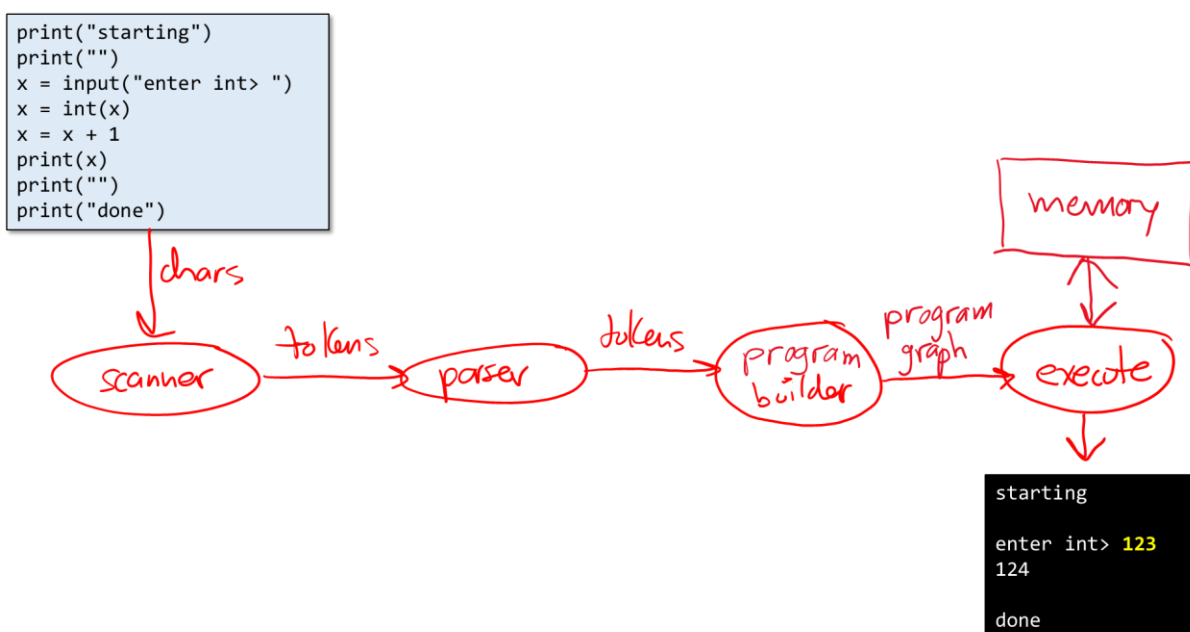
Complete By: Friday January 19th @ 11:59pm CST

Late submissions: see syllabus for late policy... No submissions accepted after Sunday 01/21 @ 11:59pm

Pre-requisites: HW 02 --- in particular chapter 3.

Overview

In project 01 you built a scanner for a subset of the Python language called **nuPython**. This scanner is used by the **parser** to check a nuPython program for proper syntax. Assuming the program is legal, an **abstract representation** of the program is built (called a “program graph”). The parser and program builder are provided, your job here in project 02 is to start building an execution environment for nuPython programs. In short, given a program graph and a memory data structure, you’ll traverse the program graph and execute the program statement by statement. Here’s an overview of the components involved in project 02, your job is implementing the right-most “execute” component:



Background...

There are two critical concepts to understand before trying to work on this assignment: what constitutes a legal nuPython program, and what does the abstract representation look like?

The nuPython language is defined using **BNF**, or Backus Naur Form. You'll learn more about this in CS 212, but in short BNF defines a series of rules for the various components of a language. For example, in nuPython, a program consists of one or more statements:

```
<program> ::= <stmts> EOS
<stmts>   ::= <stmt> [<stmts>]
```

A program is defined as <stmts> followed by EOS (end of stream). The [...] notation means optional, so <stmts> is defined as a single <stmt> followed optionally by more <stmts>. There are 5 types of statements in nuPython: assignment, function call, if then else, while, and pass. Here's the complete BNF, which we will discuss further in class:

```
<program> ::= <stmts> EOS
<stmts>   ::= <stmt> [<stmts>]

<stmt>    ::= <assignment>
            | <function_call>
            | <if_then_else>
            | <while_loop>
            | pass

<assignment> ::= ['*'] IDENTIFIER '=' <value>
<function_call> ::= IDENTIFIER '(' [<element>] ')'
<while_loop>   ::= while <expr> ':' <body>
<if_then_else> ::= if <expr> ':' <body> [<else>]
<else>         ::= elif <expr> ':' <body> [<else>]
                | else ':' <body>
<body>        ::= '{' <stmts> '}'

<value>       ::= <function_call>
                | <expr>
<expr>        ::= <unary_expr> [<op> <unary_expr>]
<unary_expr>  ::= '*' IDENTIFIER
                | '&' IDENTIFIER
                | '+' [IDENTIFIER | INT_LITERAL | REAL_LITERAL]
                | '-' [IDENTIFIER | INT_LITERAL | REAL_LITERAL]
                | <element>
<element>     ::= IDENTIFIER
                | INT_LITERAL
                | REAL_LITERAL
                | STR_LITERAL
                | True
                | False
                | None

<op> ::= '+'
```

```

| '-'
| '*'
| '**'
| '%'
| '/'
| '=='
| '!='
| '<'
| '<='
| '>'
| '>='
| 'is'
| 'in'

```

Based on these rules, the following is a legal nuPython program consisting of a function call, 3 assignment statements, and 2 more function calls:

```

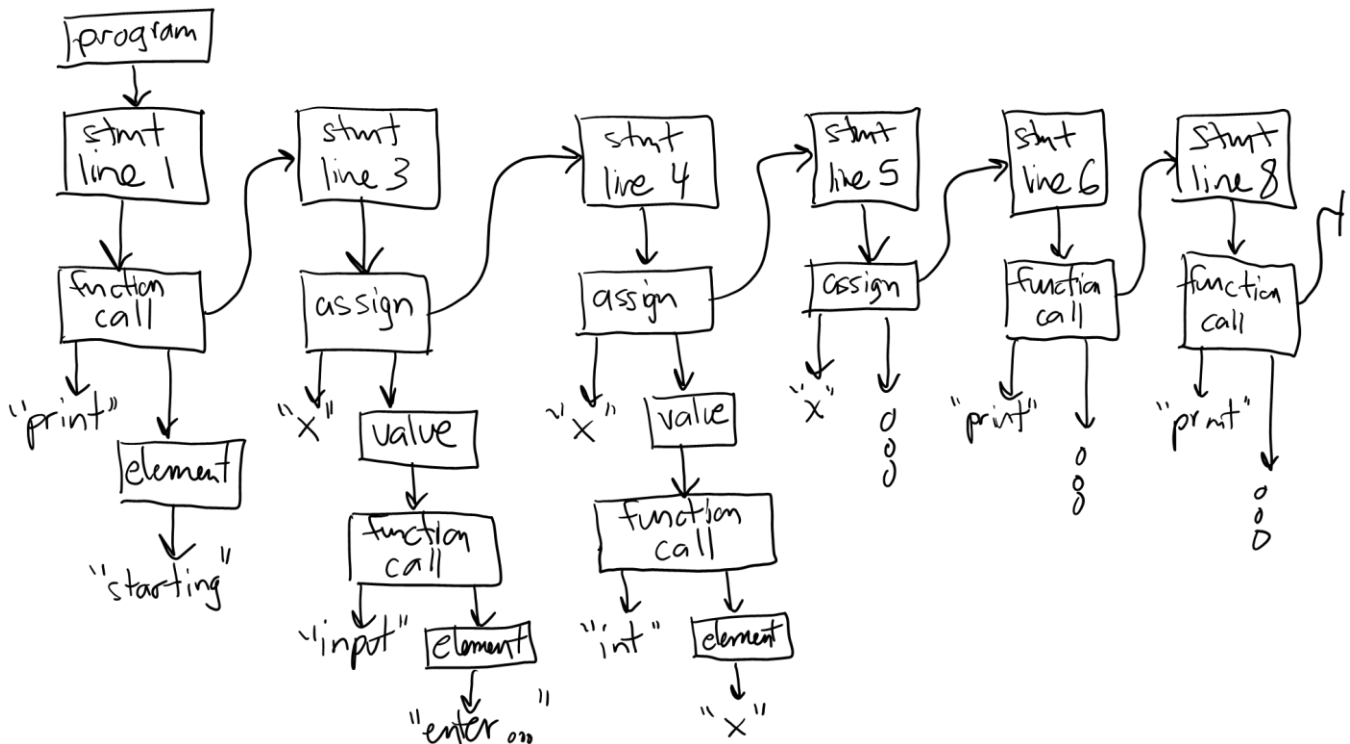
print("starting")

x = input("enter int> ")
x = int(x)
x = x + 1
print(x)

print("done")

```

Abstractly, we represent this program as a graph of STATEMENT nodes, where each node contains one or more pointers to the various components of that statement (e.g. the name of the function being called, or the name of the variable being assigned to). The above program is represented by the following graph:



<< continued on next page >>

We are providing the code to check for syntax errors (parser) and build the representation (program builder). Your job here in project 02 is to learn the representation, how to traverse, how to extract the necessary information, and then execute the nuPython program denoted by this representation.

Copying the provided files

You can work on the EECS computers or replit; if you worked in a different environment on project 01, it may be difficult to do this in project 02 since we are providing pre-compiled object code (compiler.o) that may not run in your environment. You are welcome to download the files from EECS / replit and try.

If you would prefer to work in replit, open up the CS 211 team, open "Project 02", and skip to the next paragraph... If you prefer to work on the EECS computers using VS Code, use terminal / git bash to login to **moore.wot.eecs.northwestern.edu** (or one of the other computers such as *batgirl.eecs.northwestern.edu*). Copy over the provided files to your own account as follows:

- | | |
|--|--|
| 1. Make a directory for project 01 | <code>mkdir project02</code> |
| 2. Make this directory private | <code>chmod 700 project02</code> |
| 3. Move ("change") into this directory | <code>cd project02</code> |
| 4. Copy the provided files --- the . is needed | <code>cp -r /home/cs211/w2024/project02/release .</code> |
| 5. List the contents of the directory | <code>ls</code> |

As this point you should see the directory "release" shown. Now

- | | |
|---------------------------------------|-------------------------|
| 6. Move ("change") into release dir | <code>cd release</code> |
| 7. List the contents of the directory | <code>ls</code> |

At this point you should either be logged into replit, or logged into one of the EECS computers. The next step is to copy a working version of "scanner.c" from Project 01 --- you can use your own solution, or download ours (which will become available Monday January 15th @ 12:01am). If you prefer to use our solution, you can grab a copy as follows:

EECS: `cp /home/cs211/w2024/project01/solution/scanner.c .`

Replit: download from [dropbox](#), then upload to project 02 replit

If you prefer to use your solution, you can copy as follows:

EECS: `cp ~/project01/release/scanner.c .`

Replit: download from your project 01 replit, then upload to project 02 replit

At this point you should have a copy of the provided files + a working version of "scanner.c".

<< continued on next page >>

If all is well, you should see the following files (on EECS computers type “ls” to list your files):

compiler.o	parser.h	token.h
execute.c	programgraph.h	tokenqueue.h
execute.h	ram.h	util.h
main.c	scanner.h	
makefile	scanner.c	

There are 13 provided files, here’s a summary:

C source files:	execute.c main.c, scanner.c
Header files:	execute.h, parser.h, programgraph.h, ram.h, scanner.h, token.h, tokenqueue.h, util.h
Pre-compiled code:	compiler.o
Make file:	makefile

Your job will be to modify 2 of these files: “main.c” and “execute.c”. The other files should not be modified.
[What is “compiler.o”? Some of our implementation code is pre-compiled and available only in “object” form -- this is machine code that is not human-readable. This is intentional because you may be asked to write some of this code in future projects.]

Getting started

You are given the main() function from project 01. Delete / comment out all scanner-related code: the variable declarations (lineNumber, etc.), the call to scanner_init(), and the while loop / printf / scanner_nextToken calls. Build and run your program --- it should prompt for input and then end:

```
hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
hummel> |
```

Now we want to integrate the provided components that handle the parsing (i.e. syntax checking) and building of the program graph. At the top of “main.c”, add #include statements for the following files (in this order):

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

Open “parser.h” in your editor, and you’ll see the declaration of two functions: *parser_init()*, and *parser_parse()*. Back in “main.c”, modify your main() function to call both of these functions, adding the new code as follows:

```

int main(int argc, char* argv[])
{
    .
    .
    .

    if (keyboardInput) // prompt the user if appropriate:
    {
        printf("nuPython input (enter $ when you're done)>\n");
    }
}

```

existing
code

```

//
// call parser to check program syntax:
//
parser_init();

struct TokenQueue* tokens = parser_parse(input);

if (tokens == NULL)
{
    //
    // program has a syntax error, error msg already output:
    //
}
else
{
    //
    // parsing successful, now build program graph:
    //
    printf("**no syntax errors...\n");
    printf("**building program graph...\n");
}

```

new
code

Build and run, entering a simple program as input:

```

hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
print("hi")
$
**no syntax errors...
**building program graph...
hummel>

```

Note that if the input contains a syntax error, the parser outputs an error message and the program ends (much like a real Python system). For example:

```

hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
print("hi"
$
**SYNTAX ERROR: expecting ), found '$' @ (2, 1)
hummel>

```

Now open “programgraph.h” and look for the public functions declared at the bottom. Back in “main.c”, modify the “else” block in the code you just wrote to call this function, saving the returned pointer in a variable named **program**. Then call `programgraph_print()` to print the program to the console. Build and run, and enter a simple program as input:

```

hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
# comment
print(x)
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2:
3: print(x)
4: $
**END PRINT**
hummel>

```

The `main()` function now has the program graph built that you need to start executing the nuPython program. However, to execute a nuPython program you need a data structure to simulate memory --- i.e. to hold variables and their values (such as the `x = 123` in the previous screenshot). Open the file “ram.h” and look over the provided functions; you’ll use these functions to simulate the reading and writing of memory when executing a nuPython program. In particular, look at the declarations of `ram_init()` and `ram_print()`.

Back in “main.c”, after the call to `programgraph_print()` in the “else” block, `printf` the string “**executing...\n”. Then call `ram_init()`, saving the returned pointer in a variable named **memory**. Next, call the `execute()` function declared in “execute.h” --- passing `program` and `memory` as parameters. Finally, `printf` the string “**done\n”, and call `ram_print(memory)`. All of this code should be contained in the “else” block. Build, run and test:

<< see screenshot on next page >>

```

hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
# comment
print(x)
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2:
3: print(x)
4: $
**END PRINT**
**executing...

execute(): TODO

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

```

You are ready to start executing nuPython programs!

Executing nuPython programs (part 01)

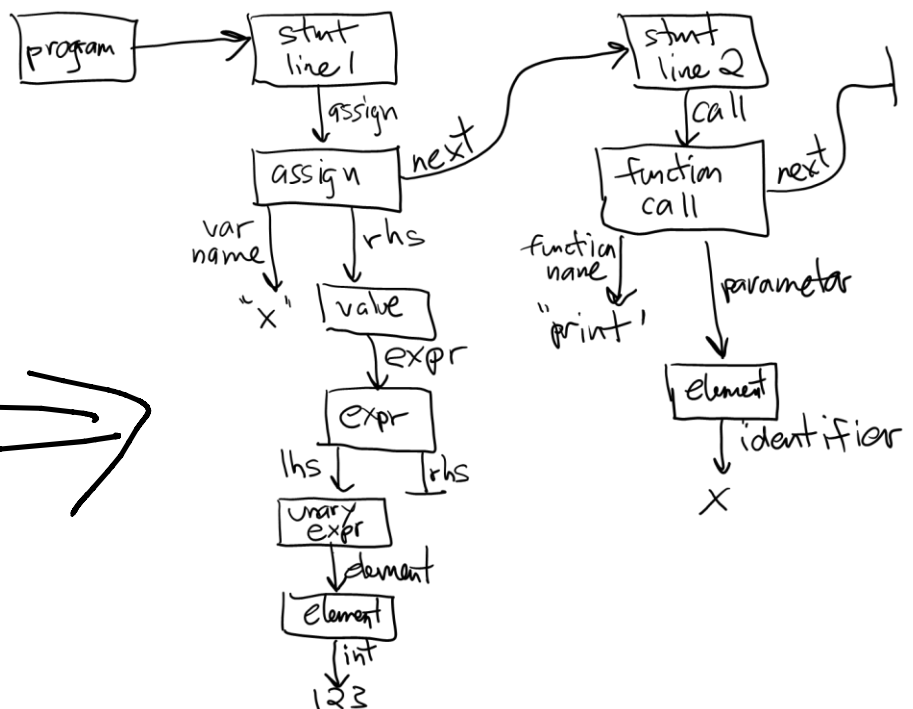
At this point all your work should be contained in “execute.c”, which is responsible for executing nuPython programs. You’ll also need to become very familiar with the contents of “programgraph.h”, which defines the program representation you’ll be working with. Be ready, this is not a trivial data structure since it must be capable of representing a large set of Python programs.

Let’s start with a high-level understanding of what we saw earlier. A nuPython program is a graph (think list for now) of STATEMENT nodes, one per statement. Consider the following two-line program, which has the program graph representation shown on the right:

```

x = 123
print(x)

```



Looking at “programgraph.h”, here’s the definition of a STMT:

```
struct STMT
{
    //
    // what kind of stmt do we have?
    //
    int stmt_type;    // enum STMT_TYPES
    int line;         // what line # does it start on?

    //
    // pointer to that stmt struct:
    //
    union
    {
        struct STMT_ASSIGNMENT* assignment;
        struct STMT_FUNCTION_CALL* function_call;
        struct STMT_IF_THEN_ELSE* if_then_else;
        struct STMT_WHILE_LOOP* while_loop;
        struct STMT_PASS* pass;
    } types;
};
```

There are 5 types of statements, defined by this enumeration:

```
//
// nuPython statement types:
//
enum STMT_TYPES
{
    STMT_ASSIGNMENT = 0,
    STMT_FUNCTION_CALL,
    STMT_IF_THEN_ELSE,
    STMT_WHILE_LOOP,
    STMT_PASS
};
```

Suppose **program** is a variable that points to the first statement in the program graph. We can determine the type of statement as follows (go ahead and type this code into the execute() function in “execute.c”):

```
struct STMT* stmt = program;

if (stmt->stmt_type == STMT_ASSIGNMENT) {
}
else if (stmt->stmt_type == STMT_FUNCTION_CALL) {
}
else if ...
```

Let’s assume **stmt** points to an assignment statement (such as “x = 123”). Let’s print out what line this stmt is on, the name of the variable being assigned to, and then let’s advance stmt to the next statement in the

program. Expanding on the earlier if-then-else statement (and continuing to modify “execute.c”):

```
if (stmt->stmt_type == STMT_ASSIGNMENT) {
    int line = stmt->line;
    char* var_name = stmt->types.assignment->var_name;

    printf("%d: %s = ...\n", line, var_name);

    stmt = stmt->types.assignment->next_stmt; // advance to next statement
}
```

A STMT node can point to 5 different types of statements using the **types** union in struct STMT. In the code above we know it’s an assignment statement so we access the details of the assignment statement via the pointer **stmt->types.assignment**, for example

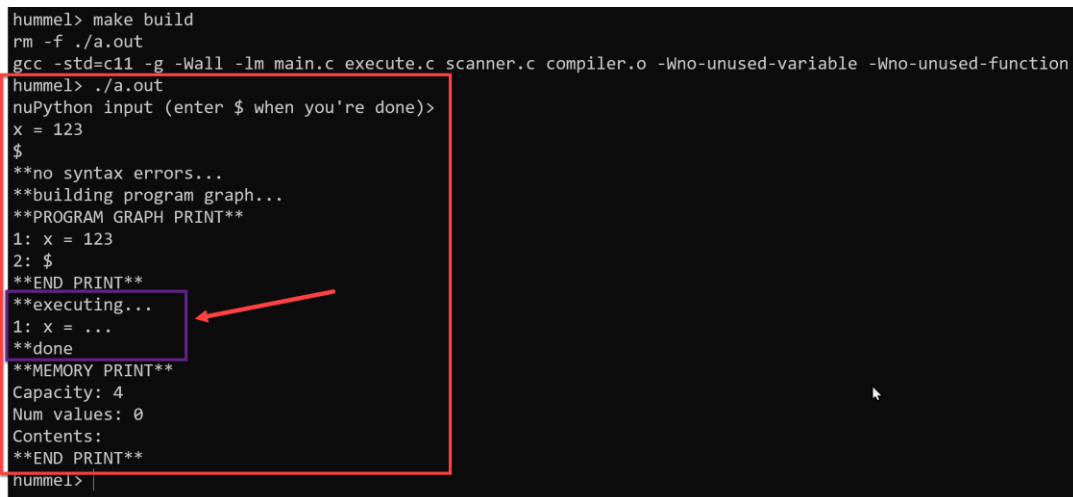
```
char* var_name = stmt->types.assignment->var_name;
```

The definition of an assignment statement can be found later in “programgraph.h”:

```
struct STMT_ASSIGNMENT
{
    // Examples:  x = 123
    //             *p = x + y
    //
    char* var_name;
    bool  isPtrDeref;
    struct VALUE* rhs; // rhs = "right-hand side"

    struct STMT* next_stmt;
};
```

Save your changes to “execute.c”, build the program, run, enter the following one-line test program, and you should get the following output:



```
hummel> make build
rm -f ./a.out
gcc -std=c11 -g -Wall -lm main.c execute.c scanner.c compiler.o -Wno-unused-variable -Wno-unused-function
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2: $
**END PRINT**
**executing...
1: x = ...
**done
**MEMORY PRINT**
Capacity: 4
Num values: 0
Contents:
**END PRINT**
nummel>
```

Of course, the meaning of an assignment statement is to assign a value into memory, so executing a

statement such as “x = 123” should be updating memory, not printing to the screen. The next section will get you started executing nuPython statements...

Executing nuPython programs (part 02)

1. There are only 3 types of statements you need to worry about in project 02: assignment, function call, and pass. Ignore if statements and while loops.
2. Modify “execute.c” to traverse through the statements in the given program, outputting the line # and the type of statement --- the output format doesn’t matter. Build upon the code discussed in the previous section:

```
struct STMT* stmt = program;

while (stmt != NULL) { // traverse through the program statements:

    if (stmt->stmt_type == STMT_ASSIGNMENT) {

        stmt = stmt->types.assignment->next_stmt; // advance
    }
    else if (stmt->stmt_type == STMT_FUNCTION_CALL) {

        stmt = ...
    }
    else {
        assert(stmt->stmt_type == STMT_PASS);

        stmt = ...
    }
} //while
```

Build, run, and test --- input a program with at least one statement of each type.

3. What does it mean to execute a **pass** statement? In Python “pass” means do nothing, and is used to denote an empty loop or if-then-else case. Comment out the print statement, and mark “pass” as done.
4. Add a helper function named *execute_function_call()* which is passed pointers to a statement and the memory. Define the function to return true if the statement executes successfully, and false if not. Assume for now that all function calls are of the form **print()** --- no parameter --- or **print(...)** with a string literal parameter. Print with no parameter prints a newline character ‘\n’, print with a string literal prints the literal followed by a newline character. Implement *execute_function_call()* to handle these two cases with calls to *printf()*, and return true.
5. Modify the *execute()* function to call *execute_function_call()* when the statement is a function call. Build your program, run, and test with input programs like this:

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

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

The output for this program should look like this:

```
nuPython input (enter $ when you're done)>
print("starting")
print()
print('done')
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: print('starting')
2: print()
3: print('done')
4: $
**END PRINT**
**executing...
starting

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

6. Add a helper function named `execute_assignment()` which is passed pointers to a statement and the memory. Define the function to return true if the statement executes successfully, and false if not. Assume for now that all assignment statements are of the form “variable = int_literal”, for example “x=123”. Implement `execute_assignment()` to handle this case. [*Suggestion: look back at the diagram on the bottom of page 8.*]

Think about what it means to execute an assignment statement... In this case the integer literal, which is a string, must be converted to an integer --- use C’s `atoi()` function. [Here’s a good online [reference](#) for C library.] Then store the integer into memory by calling `ram_write_cell_by_id()`. See “ram.h” for details... To call `ram_write_cell_by_id()`, you need to define a struct `RAM_VALUE` variable. Set this variable’s `value_type` to `RAM_TYPE_INT`, and set `types.i` to the result of calling `atoi()` on the literal.

7. Modify the `execute()` function to call `execute_assignment()` when the statement is an assignment. Build your program, run, and test with input programs like this:

```
x = 123
y = 456
```

The output for this program should look like this:

<< continued on next page >>

```

nuPython input (enter $ when you're done)>
x = 123
y = 456
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2: y = 456
3: $
**END PRINT**
**executing...
**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
Contents:
  0: x, int, 123
  1: y, int, 456
**END PRINT**

```

Given the input program in step 1, notice that its execution in step 2 produces no output --- because the program contains no `print()` statements. However, we can see if the assignments were executed correctly by looking at the contents of memory when the program ends --- notice that `x` and `y` are properly output as integers with the values 123 and 456, respectively.

8. Extend `execute_function_call()` to handle the case of calls to `print` with an integer literal, such as `print(123)`. In the case of an integer literal, convert the literal to an integer using `atoi()`, then `printf` using `%d`. Build, run and test.
9. Extend `execute_function_call()` to handle the case of calls to `print` with an identifier, such as `print(x)`. Call the function `ram_read_cell_by_id()` to retrieve the value of the identifier; you may assume that all values returned are integers, and for the moment assume the identifier exists in memory. See “`ram.h`” for details. Build, run and test. Here’s an example --->

```

nuPython input (enter $ when you're done)>
print("starting")
print(99)
x = 123
y = 456
print(x)
print(y)
print('done')
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: print('starting')
2: print(99)
3: x = 123
4: y = 456
5: print(x)
6: print(y)
7: print('done')
8: $
**END PRINT**
**executing...
starting
99
123
456
done
**done
**MEMORY PRINT**
Capacity: 4
Num values: 2
Contents:
  0: x, int, 123
  1: y, int, 456
**END PRINT**

```

10. What if the identifier does not exist in memory? What does `ram_read_cell_by_id()` return? This is a *semantic* error, i.e. the program's *meaning* is undefined. In this case print an error message in the format shown below and return false from `execute_function_call()`:

```
nuPython input (enter $ when you're done)>
x = 123
print(y)
y = 456
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2: print(y)
3: y = 456
4: $
**END PRINT**
**executing...
**SEMANTIC ERROR: name 'y' is not defined (line 2)
**done
**MEMORY PRINT**
Capacity: 4
Num values: 1
Contents:
  0: x, int, 123
**END PRINT**
```

Notice that when a semantic error occurs, program execution stops --- in the example above, the statement “`y = 456`” was never executed (look at the contents of memory). This implies you need to modify `execute()` to evaluate the return value from calling `execute_function_call()`. If false is returned, then `execute()` needs to immediately return back to `main()`. [While you are here, you might as well apply the same logic to calling `execute_assignment()`.] Build, run and test.

11. Okay, last step. Add support for assignment statements involving binary expressions on the right-hand side, for example “`x = y + 10`”. Support 6 operators: `+`, `-`, `*`, `/`, `%` and `**`. Also support the presence of integer literals or identifiers on either side of the operator. Examples include:

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

Don't try to solve this all at once, solve in steps. And as you solve, this is a great time to think about the use of helper functions to minimize code duplication. [*Hint: can you write a single helper function that given a lhs or a rhs, returns the value as a RAM_VALUE? Then you just apply the operator to the two values and write the result.*] Handle semantic errors involving undefined variables.

Grading and Electronic Submission

You will submit your “main.c” and “execute.c” files to [Gradescope](#) for grading. If you are working on replit, download these files to your laptop and then drag-drop to submit to gradescope. If you are working on an EECS computer, do the following:

1. Open a terminal window
2. Change to your release directory
3. make submit

Keep in mind that Gradescope is a grading system, not a testing system. The idea is to give you a chance to improve your grade by allowing you to resubmit and fix errors you may have missed --- Gradescope is not meant to alleviate you from the responsibility of testing your work. For this reason, (1) Gradescope will not open until 2-3 days before the due date, and (2) submissions to Gradescope are limited to **4 submissions per 24-hour period**. A 24-hour period starts at midnight, and after 4 submissions, you cannot submit again until the following midnight; unused submissions do not carry over, you get exactly 4 per 24-hour period (including late days).

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your “Submission History”. This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements:

1. *You must update the header comment at the top of “main.c”. Update the description, and add your name.*
2. *You must provide a header comment at the top of “execute.c”, replacing the << WHAT... >> with meaningful text, your name, etc.*
3. *Definition and use of function `execute_function_call()`, with header comment.*
4. *Definition and use of function `execute_assignment()`, with header comment.*
5. *The definition of at least one additional helper function, with header comment, to aid in the implementation of `execute_assignment()` in the case of binary expressions. Your helper function must be non-trivial, no one-line functions that just print something; your helper function must contain at least 10 lines of meaningful code.*

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found [here](#). In summary, here are NU’s eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*

4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity [website](#). With regards to CS 211, unless stated otherwise, all work submitted for grading **must** be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.

