CS 211
Fundamentals of Computer Programming II
Winter 2023

Northwestern University

Project #03 (v1.1)

Assignment: nuPython program execution (part 02)
Submission: Gradescope
Policy: individual work only, late work is accepted
Complete By: Friday January 26th @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 01/28 @ 11:59pm

Pre-requisites: Careful review of project 02 solution (when available)

Overview

In project 02 you laid the foundation for executing **nuPython** programs, implementing the ability to perform basic computation and print results. For example:

```
print('starting')

x = 10
y = x ** 3
print(x)
print(y)

print('done')
```

Here in project 03 you're build upon this foundation to add more data types (real numbers, strings, booleans), more functions (input, int, float), relational operators, string concatenation, and while loops. For example:

```
s = input('Enter an integer> ')
N = int(s)
result = "project03:"

i = 1
while i <= N:
{
   result = result + "a"
   print(result)
   i = i + 1
}
```

```
**executing...
Enter an integer> 5
project03:a
project03:aa
project03:aaa
project03:aaaa
project03:aaaaa
**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
 0: s, str, '5'
 1: N, int, 5
 2: result, str, 'project03:aaaaa'
 3: i, int, 6
**END PRINT**
```

Reference

For reference, here's the definition of our **nuPython** language, in Backus Naur Form:

```
<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
```

Copying the provided files

A few small changes were made for project 03, in particular to the "programgraph.h" and "compiler.o" files. For this reason we'll start with a fresh set of files for project 03. You'll also need your solution to project 02: "main.c" and "execute.c". If you prefer, you can use our solution which will be made available @ 12:01am on Monday January 22nd (i.e. after the late deadline for project 02). As before we'll make files available on the EECS computers, replit, and dropbox. Details below.

If you prefer to work on replit, login to replit.com and open "Project 03". You will need to upload your "main.c" and "execute.c" files from Project 02. Click the run button to build and run, use the Shell to run valgrind and gdb. You should have a working solution to project 02 at this point.

If you prefer to work in your own environment which is Linux-based, you can download the necessary files from dropbox. If you are working with Windows WSL running Linux, try this version.

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 03 — `mkdir project03`
2. Make this directory private — `chmod 700 project03`
3. Move ("change") into this directory — `cd project03`
4. Copy the provided files --- the `.` is needed — `cp -r /home/cs211/w2024/project03/release .`
5. List the contents of the directory — `ls`

As this point you should see the directory "release" shown. Let's confirm, and then copy over your solution from project 02:

6. Move ("change") into release dir — `cd release`
7. List the contents of the directory — `ls`
8. Copy your project 02 solution files: — `cp ~/project02/release/main.c .` and `cp ~/project02/release/execute.c .`

At this point you should have a working solution to project 02 (build and run to make sure).

Our solution

Our solution will be made available @ 12:01am on Monday January 22nd (i.e. immediately after the late deadline for project 02). You can download our "main.c" and "solution.c" files from this dropbox link, or you can copy them on the EECS computers as follows (use the "mv" command to rename the files after copying):

```
cp /home/cs211/w2024/project02/solution/main.c  main-solution.c

cp /home/cs211/w2024/project02/solution/execute.c  execute-solution.c
```

Assignment details

A few preliminary notes before you start:

a) You do not have to worry about memory leaks --- i.e. you do NOT have to free memory here in project 03. We will address the proper freeing of memory in project 04. For this reason, we would recommend that you do NOT call the *ram_free_value( )* function declared in "ram.h". This function has subtle behavior that may cause errors until you fully understand its purpose. Better to avoid this function until project 04.

b) The purpose of enumerated values like *STMT_ASSIGNMENT* and *RAM_TYPE_STR* is to make code easier to read. Using their numeric equivalents such as 0 and 2 is considered a very bad practice. To prevent you from following this bad practice, we are going to randomly renumber all enumerated values in the .h files when you submit to Gradescope. This will not present a problem if you use the names (e.g. STMT_ASSIGNMENT); this will cause errors if you use numeric values (e.g. 0 or 2).

c) Turns out we spelled OPERATOR_ASTERISK incorrectly in "programgraph.h" --- this has been fixed.

d) How you approach this project will largely determine the difficulty. Good function design is critical here; the more you copy-paste, the more difficult things will be. You might want to review our solution (when available) to see how we used functions to decompose the problem into reusable / extensible components.

When you're ready, here are the components of nuPython that you need to implement here in Project 03. We recommend approaching in these steps:

1. **print(literal)** should now support literals of type int, real, string, and booleans True and False (printed as True and False).

2. **x = literal** should now support literals of type int, real, string, and booleans True and False. Note that booleans are stored in memory as type RAM_TYPE_BOOLEAN, where False has the integer value 0 and True has the integer value 1.

3. **print(var)** where var can be of RAM_TYPE int, real, string or boolean. When printing a boolean integer value, print False if the value is 0 and otherwise print True.

4. At this point, an example input is shown on the right. Note that the program graph is not printed to reduce the size of the screenshot; your main( ) function should continue to output the program graph.

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
print(123.45)
print(True)
print(False)
x = True
y = False
s = 'this is a string'
d = 3.14159
print(x)
print(y)
print(s)
print(d)
$
**no syntax errors...
**building program graph...
**executing...
123.450000
True
False
True
False
this is a string
3.141590
**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
 0: x, boolean, True
 1: y, boolean, False
 2: s, str, 'this is a string'
 3: d, real, 3.141590
**END PRINT**
hummel>
```

5. Extend the **operators** +, -, *, **, / and % to work for integers, reals, and strings, under the following conditions:
    a. If both operands are integer, the result is an integer.
    b. If both operands are real, the result is a real.
    c. If one operand is an integer and the other is a real, perform the computation using reals and store the result as a real.
    d. If both operands are strings and the operator is +, perform string concatenation.
    e. All other combinations are illegal and should result in a semantic error as shown below.

    Note that % does not work for real values, use *fmod( )* instead. When performing string concatenation, you will need to dynamically allocate enough memory for the concatenated result; don't forget to malloc an extra character for the \0 that you must store at the end of the new string.

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 3.14159
y = x - 6.2
z = 1 + y
s = "start of string"
s2 = s + " end of string"
a = x
b = x ** 2.2
print(x)
print(y)
print(z)
print(s2)
print(b)
$
**no syntax errors...
**building program graph...
**executing...
3.141590
-3.058410
-2.058410
start of string end of string
12.408775
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 3.141590
 1: y, real, -3.058410
 2: z, real, -2.058410
 3: s, str, 'start of string'
 4: s2, str, 'start of string end of string'
 5: a, real, 3.141590
 6: b, real, 12.408775
**END PRINT**
hummel>
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
y = x ** 3.1
z = "a string" + 1
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: invalid operand types (line 3)
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = True
y = False
z = x + y
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: invalid operand types (line 3)
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
y = 456
z = x * yy
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: name 'yy' is not defined (line 3)
```

6. Add support for the **relational operators** ==, !=, <, <=, >, and >=, under the following conditions:
    a. If both operands are integer, compare as integers.
    b. If both operands are real, compare as reals.
    c. If one operand is an integer and the other is a real, compare both as real.
    d. If both operands are strings, perform a case-sensitive compare using *strcmp( )*. Note that strcmp(s1, s2) returns a negative number if s1 < s2, 0 if s1 = s2, and a positive number if s1 > s2.
    e. All other combinations are illegal and should result in a semantic error as shown earlier.

    The result is always a value of type RAM_TYPE_BOOLEAN; store the integer value 0 if the result if False and 1 if the result is True.

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 10 < 20
y = 3.14 > 2.0
z = "apple" <= "banana"
b = "apple" != "apple"
print(x)
print(y)
print(z)
print(b)
$
**no syntax errors...
**building program graph...
**executing...
True
True
True
False
**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
 0: x, boolean, True
 1: y, boolean, True
 2: z, boolean, True
 3: b, boolean, False
**END PRINT**
hummel>
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 3.14 < "string"
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: invalid operand types (line 1)
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = y < 100
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: name 'y' is not defined (line 1)
```

7. Add 3 **more functions**: *input('…')*, *int(s)*, and *float(s)*. These functions appear as part of assignment statements (i.e. VALUE_FUNCTION_CALL). You may assume that *input('…')* is always passed a string literal, and *int(s)* and *float(s)* are passed identifiers of variables that (a) exist in memory and (b) have values of type RAM_TYPE_STR.

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
s = input('enter an int> ')
i = int(s)
s = input('enter a real> ')
f = float(s)
print(i)
print(f)
x = f ** i
print(x)
$
**no syntax errors...
**building program graph...
**executing...
enter an int> 3
enter a real> 4.2
3
4.200000
74.088000
**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
 0: s, str, '4.2'
 1: i, int, 3
 2: f, real, 4.200000
 3: x, real, 74.088000
**END PRINT**
hummel>
```

Python's *input('…')* function prompts the user, reads the entire line, removes the end-of-line characters, and stores the resulting string into the variable. Here's the C code to safely input a line and remove the EOL characters:

```
char line[256];

fgets(line, sizeof(line), stdin);

// delete EOL chars from input:
line[strcspn(line, "\r\n")] = '\0';
```

Depending on how you implement things, you may need to dynamically allocate memory to hold the user's input (e.g. if you plan to return the input string from a function that contains the above code). Note that "util.h" declares a function *dupString(s)* that duplicates a string for you using dynamic memory allocation; feel free to use this function.

Python's *int(s)* function converts a string to an integer, outputting an error message if the conversion fails; *float(s)* converts a string to a real. To perform these conversions in C, using *atoi( )* and *atof( )*. How can you tell if the conversion fails? Both functions atoi( ) and atof( ) return 0 if the conversion fails. However, converting the string "0" (or "00", or "000", …) should NOT be flagged as an error.

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
s = input('enter something> ')
i = int(s)
print(i)
$
**no syntax errors...
**building program graph...
**executing...
enter something> apple
**SEMANTIC ERROR: invalid string for int() (line 2)
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
s = input('enter something> ')
f = float(s)
$
**no syntax errors...
**building program graph...
**executing...
enter something> fred
**SEMANTIC ERROR: invalid string for float() (line 2)
```

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
s = input('enter> ')
i = int(s)
s = input('enter> ')
f = float(s)
print(i)
print(f)
$
**no syntax errors...
**building program graph...
**executing...
enter> 000
enter> 0000000
0
0.000000
**done
**MEMORY PRINT**
Capacity: 4
Num values: 3
Contents:
 0: s, str, '0000000'
 1: i, int, 0
 2: f, real, 0.000000
**END PRINT**
hummel>
```

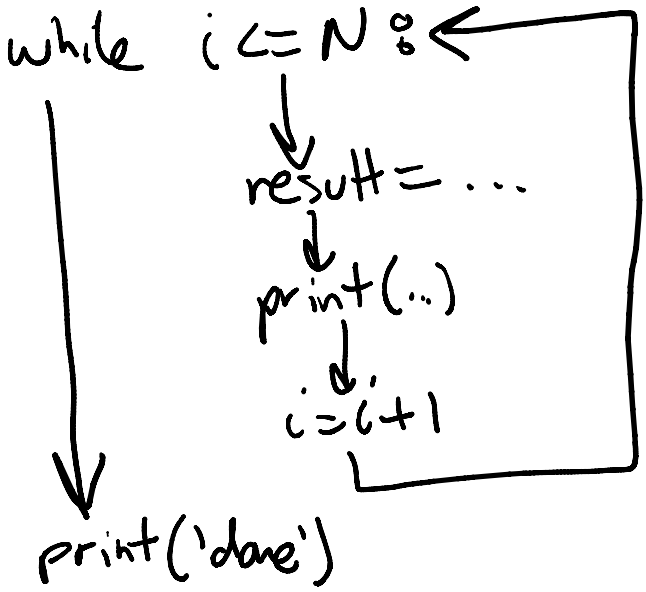
8. Last feature: add support for **while loops**. While loops can be nested (to any level), this should not present a problem. Here's an example we saw earlier:

```
s = input('Enter an integer> ')
N = int(s)
result = "project03:"

i = 1
while i <= N:
{
   result = result + "a"
   print(result)
   i = i + 1
}

print('done')
```

```
**executing...
Enter an integer> 5
project03:a
project03:aa
project03:aaa
project03:aaaa
project03:aaaaa
done
**done
```

In the program graph, a STMT_WHILE_LOOP has 2 STMT pointers: **loop_body** points to the first statement in the loop body (the assignment statement "result = …"), and **next_stmt** points to the statement that follows the loop ("print('done')").

The last statement in the loop body --- "i = i + 1" in the example above --- forms the loop by pointing back to the STMT denoting the while loop. This forms a cycle in the program graph, and this is how you can tell when you are back at the beginning of the loop (in which case you re-check the condition and decide if you should execute the loop again).



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. *Update the header comment at the top of "execute.c" and "main.c", especially if you are using our solution (add you name, update the description, etc.)*
2. *Helper functions, helper functions, helper functions. Our solution has 5 helper functions in addition to the required execute( ) function. You must provided at least 3 additional helper functions for a minimum of 8 helper functions + execute( ). All header functions must have a header comment.*
3. *Helper functions must be < 100 lines of code as counted by Gradescope, and must contain at least 5 lines of meaningful code. Trivial helper functions are not acceptable.*
4. *Monitor your Gradescope output, which will report functions > 100 lines of code. We may also impose a cyclomatic complexity limit on your helper functions to prevent the writing of overly-complex, hard to understand functions. Here's a brief overview of cyclomatic complexity.*

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.