

Project Extra-Credit (v1.0)

Assignment: **nuPython program execution with pointers**

Submission: **Gradescope**

Policy: **individual work only, late work not accepted**

Complete By: **Sunday February 11th @ 11:59pm CST**

Late submissions: this is an extra-credit project, no late submissions.

Pre-requisites: **N/A**

Overview

We are returning back to the focus of projects 02 and 03, which is the execution of **nuPython** programs. In this extra-credit project the goal is to add the concept of C-style pointers to Python. Here's an example:

```
print('starting')

x = 123      # x occupies cell 0 in memory
p = &x      # p is a pointer, containing address of x
print(p)     # prints 0

y = 456      # y will end up in cell 2 of memory
p2 = &y     # p2 is a pointer containing address of y
print(p2)    # prints 2

z = 0        # y ends up in cell 4
rp = &z     # rp is a pointer containing address of z
print(rp)    # prints 4

*rp = *p + *p2  # follow ptrs, add, follow ptr & store
print(z)        # prints 579

print('done')
```

```
**executing...
starting
0
2
4
579
done
**done
**MEMORY PRINT**
Capacity: 8
Num values: 6
Contents:
0: x, int, 123
1: p, ptr, 0
2: y, int, 456
3: p2, ptr, 2
4: z, int, 579
5: rp, ptr, 4
**END PRINT**
```

The good news is changes are fairly limited in scope: unary expressions, assignment, and the print function. There is not a lot of coding required, the assignment is more conceptual than it is about implementation.

The extra-credit project is worth 50 points, which can be used to bolster project scores (C or C++) that are < 100; no project score may exceed 100 points. These points cannot be used for any other purpose.

Assignment details

You are required to add the following five features to nuPython:

1. **Add support for the unary & operator**, which obtains the address of a variable. You may assume the & will only be used with an identifier, and only in the context of a unary expression (i.e. not part of a binary expression such as &x + &y). Example:

```
p = &x
```

After execution of the above statement, p is a variable in memory of type RAM_TYPE_PTR, with an integer value containing the address of x in memory (the RAM module has a function for obtaining the address of an identifier). If x does not exist, output a semantic error of the format

```
**SEMANTIC ERROR: name 'x' is not defined (line 1)
```

and stop execution.

2. **Modify the print() function** to handle the printing of pointer variables; output the value as an integer. Example:

```
x = 123
p = &x
print(p)      # this should print 0
```

3. **Add support for pointer arithmetic**, of a limited nature. In particular, support expressions of the form pointer variable +/- integer literal. Examples:

```
a = p + 1
b = q - 2
```

Assuming **p** and **q** are pointer variables, then **a** and **b** will also become pointer variables. Output a semantic error and stop execution if p or q are not defined.

4. **Add the unary * operator**, which dereferences a pointer to obtain the value. You may assume * will be used with an identifier, but it may appear in the context of unary or binary expressions. Examples:

```
a = *p
b = *q - 10
```

```
c = *p * *q
```

If the pointer variables does not exist (e.g. p in the code above), output a semantic error of the format

```
**SEMANTIC ERROR: name 'p' is not defined (line 1)
```

and stop execution. You also need to check that the variable's value is in fact a pointer, and not an integer, real, string, boolean or None. For example, this is a semantic error:

```
x = 0
a = *x
```

In this case output a semantic error of the format

```
**SEMANTIC ERROR: invalid operand types (line 2)
```

and stop execution. Finally, you need to confirm the pointer is valid, since pointer arithmetic may cause an address to be out of bounds. Example:

```
x = 123
p = &x
print(p)      # this should print 0

p = p - 1
y = *p        # ERROR: invalid address!
print(y)
```

In this case output a semantic error of the following format

```
**SEMANTIC ERROR: 'p' contains invalid address (line 6)
```

and stop execution.

5. **Add support for pointer-based assignment**, i.e. assigning to a memory location denoted by a pointer. Example:

```
x = 123
y = 456
z = 789
p = &x
p = p + 2      # p now points to z
print(z)       # 789
*p = *p + 1
print(z)       # 790
```

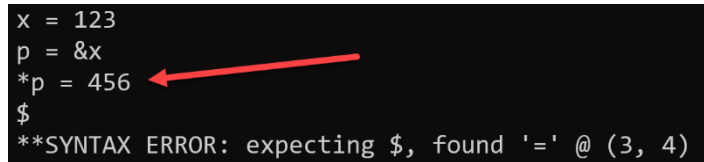
```
**executing...
789
790
**done
**MEMORY PRINT**
Capacity: 4
Num values: 4
Contents:
0: x, int, 123
1: y, int, 456
2: z, int, 790
3: p, ptr, 2
**END PRINT**
```

Check for semantic errors in the assignment, i.e. (1) does the variable exist (e.g. p), (2) does the

variable contain a pointer value (e.g. is p in fact a pointer), and (3) is the pointer value a valid memory address. The format of the semantic errors should follow the format shown earlier.

NOTE: there's an error in the provided, pre-compiled code during the parsing of pointer-based assignment statements. For example, the following code fails to compile even though it's legal:

```
x = 123
p = &x
*p = 456
```



```
x = 123
p = &x
*p = 456
$
**SYNTAX ERROR: expecting $, found '=' @ (3, 4)
```

The solution is to precede the assignment statement with a print or pass statement, e.g.

```
x = 123
p = &x

pass
*p = 456
```

Getting started

Before you start, be aware that the TAs have been instructed not to answer questions about this project, and cannot provide help nor guidance. You may post clarifying questions on Piazza, but we may or may not answer those questions. This is an extra-credit project, and you are meant to solve it completely on your own.

Revisit your project 03 release directory on the EECS servers, or your Project 03 on replit. You can work from your own solution to project 03, or you can work from the partial solution we are providing (which is sufficient for the purposes of this extra-credit assignment).

If you want to use our provided “execute.c” file... On the EECS servers, login and cd to your project03 release directory, and rename your existing work as follows:

```
mv execute.c my-execute.c
```

Then copy our file:

```
cp /home/cs211/w2024/project03/solution/execute.c .
```

If you are working on replit, use the file explorer pane to rename your existing “execute.c” file. Then download our file from [dropbox](#) and upload to replit.

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
```

Grading and Electronic Submission

You will submit your “execute.c” file 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. `/home/cs211/w2024/tools/projectxc submit execute.c`

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 50/50 points, there will be no manual review for style / commenting. However, we may manually review to confirm adherence to assignment requirements.

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 your 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.