CS531 Project #2: RPN Calculator
Due: Friday, Nov 8th at 11:59PM

This is to be an individual effort. No partners. No Internet Resources
(*See the CS Honor Code Below*)

https://cs.gmu.edu/resources/honor-code/
https://cs.gmu.edu/resources/honor-code/statement-on-academic-integrity/

Before you Start: This project requires a good understanding of basic data structures in C. Review the text and lectures for Memory and Pointers, Dynamic Memory (malloc/free), Structs and Struct Pointers, Linked Lists, **File I/O, Stacks**, and **Hash Tables**. In this project, you will be working with a singly linked list with a dummy node to implement both the stack functions and the hash table functions.

Overview

For this assignment, you are going to use C and the Stack and Hash Table data structures to implement a **Postfix-notation (RPN) Calculator.** This calculator will read in a one-line program from a file. This line read from the file will then be passed into a pre-written **tokenizer**, which will take every individual piece of the program and turn them into **tokens**. In every step of the calculator, the calculator will get the next token and you will process it by either pushing it into a **stack**, or by executing the operation the token represents. This calculator will work with both number values and variables, which will be stored as Symbols with a variable name and a value in a symbol table (**hash table**).

Postfix-Notation in Math

In math, postfix-notation is a way of writing an equation where the operands come before the operator. For example, **5 10 +** is postfix-notation for 5 + 10. Since the operands come before the operator, when you reach an operator, you can just perform the operation on the two prior values.

Postfix	Normal (Infix) Equation	Result
5 10 +	5 + 10	15
2 5 -	2 - 5	-3
10 5 /	10 / 5	2
2 3 *	2 * 3	6

Operators in this notation can also be chained together to allow for larger expressions.

5 10 + 2 * 4 2 / +

These expressions are evaluated from left-to-right. We start with **5 10 +**, when we see the + operator, we add the two preceding operands, 5 and 10, together and get 15. Now we have **15 2 ***, using the previous result. This gives us 30. After the 30, we see **4 2 /**, so we perform that division on the preceding two operands, to get 2. Now we have **30 2 +** and get our result of 32. Notice how you can have a complicated expression in postfix notation without having to use any parenthesis.

This expression is equivalent to the following expression in infix notation, which needs parentheses.

((5 + 10) * 2) + (4 / 2)

Calculator Overview
Your calculator will process the following **types** of **tokens** that may be in a program.

Value	A value is just a number in the expression, like **5** or **-2**
Variable	A variable is a name that is in an expression, like **foo** or **sum**
Operator	An operator is a mathematical operator. We will have: **+**, **-**, **/**, and *****
Assignment	This is a special token to assign a value to a variable. This is an **=** in the equation
Print	This token tells your calculator to **print** out the current stack top.

When you start the calculator, you will see the current program that was read in from the file. As you execute each step of the program, you will see the remaining tokens at any step.

```
.------------------
| Program Step =  0
|-----Program Remaining
| x 3 2 + = foo 4 = x foo + print
o------------------
```

This represents a program that was just read in from a file and is starting its execution. At each step of the execution, the calculator will get the next token from the program and process it according to its type. This will continue until all tokens have been processed.

Rules for Processing each Token:
Here are the basic rules for how to process each token by their type as you get them:

Value	**Push** the token on to the stack.
Variable	**Push** the token on to the stack.
Operator	**Pop two** tokens off the stack, perform the operation on them, **push** the result.
Assignment	**Pop two** tokens off the stack (value and variable), **put** it in the symbol table.
Print	**Pop one** token off the stack and **print** it to the screen as output.

For the example program at the top of this page, we get each token one by one and pass it in to your functions. The first step would get the first token, **x**, and push it on to the stack. You would then get the next token, **3**, and push it on the stack. Then you get the next token, **2**, and push it on the stack.

```
.------------------
| Program Step =  3
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 2 3 x
|-----Program Remaining
| + = foo 4 = x foo + print
o------------------
```

Here, you can see the three tokens (**2 3 x**) that have been pushed on to the stack, as well as the remaining program tokens below it. The next step will get the **+** token. At this point, the rule says you will pop two tokens from the stack, **2** and **3**, then add them together (**2 + 3**) and get **5**. This resulting token then gets pushed on to the stack, as shown in the output on the next page.

```
.------------------
| Program Step =  4
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 5 x
|-----Program Remaining
| = foo 4 = x foo + print
o------------------
```

The next token is an **=**, so this means you pop the **5** off as the value to assign into a variable, and then you pop the **x** from the stack as the name of the variable. You will then set the value for variable **x** to **5** in your Symbol Table (hash table).

Continuing with this example, **foo** would be assigned the value of **4**, then the values of **x** and **foo** are added together (**+** token). The result of all of this (**9**) will be on the stack at the end when the **print** token is reached. This will then be the output of the calculator.

Project Details

In this project, you will be working with a large number of files that contain code that is already implemented, and **three files** that you need to finish implementing in order to make this calculator work. For this project, you will be writing code in three files only:

rpn.c	The main calculator logic code to run the programs and perform the file I/O.
stack.c	The stack implementation. Singly linked-list with a dummy node.
hash.c	The symbol table implementation. Separate Chaining with Rehashing.

The other files are also important to examine as you write these.

stack.h	Definitions for your **Stack_head** structure. (Your dummy node type)
node.h	Definitions for your **Node** structure used in your stack.
node.c	Functions you can use to work with **Node**s (eg. **node_create** and **node_free**)
symbol.h	Definitions for your **Symtab** structure used for your hash table. (its Header) Definitions for **Symbol** structure used as the nodes for your separate chaining.
symbol.c	Functions you can use to work with **Symbol**s (eg. **symbol_copy**, **symbol_free**, and **symbol_create**)
token.h	Definitions for the **Token** structure you will be working with in processing. This also has the definitions for your operators and types to compare against. For example, if token->type == TYPE_VALUE, then it's a value. If the type is TYPE_OPERATOR, then if token->oper == OPERATOR_PLUS, it's a **+** token.
token.c	Functions you can use on **Token**s (eg. **token_create_value** and **token_free**)

Token Structure Description

A Token struct contains the following fields:

type	This will be one of the 5 TYPE_ constants at the top of **token.h**
oper	If type == TYPE_OPERATOR, then this will be one of the 4 OPERATOR_ constants to let you know which operator it is.
variable	This will have the name of a TYPE_VARIABLE token. (eg. **x** or **foo**)
value	This will have the value of a TYPE_VALUE token. (eg. **5** or **-1**)

Overview of Implementation

In **rpn.c**, the main loop is already written for you (rpn function). This starts by calling **read_file**, which you will be writing to read the passed in filename into the passed in line (up to MAX_LINE_LEN) in size. After this, it will automatically pass this line into the tokenizer and get each token from the program, one by one. Notice there is code in here to print out different parts of the program for convenience.

You will mainly be writing the **parse_token** function. This function passes you the pointer to the Symbol Table (Hash table), and a pointer to the Stack's dummy node. You also get the next **Token** to process. You will write this function to implement the rules for processing the token. For example, if the Token (**tok**) you are passed in has type == TYPE_VARIABLE, then you will need to push it on to the stack. If it is TYPE_OPERATOR, then you need to pop two tokens from the stack, perform the operation, then push the result back to the stack.

Working with Tokens

Your stack will be set up to push tokens (which you will store in **Node**s of your linked list) on to the stack and pop tokens off of the stack natively. When you perform an operation on two tokens, such as **+**, you will get a resulting value that needs to be pushed back on to the stack. In this case, you should free the two tokens you popped off earlier, then use the **token_create_value** function from token.c to create a brand new token with the result, so you can push it back on to the stack.

In general, when you no longer need a token anymore, you should use the **token_free** function from token.c to free the token.

Stack Implementation

You will also be implementing all of the functions in **stack.c** to create your Linked-List implementation of a Stack data structure. The header structure for the Stack is in **stack.h** and has these members:

count This needs to have the number of items in your stack. It starts at 0.
top This is the pointer to the top (front) of your Linked List implementation of the Stack.

The struct definition for your **Node**s is in **node.h** and has the following members:

tok This is a pointer to the **Token** struct you will be storing on the Stack.
next This is just part of the normal singly-linked list.

You will start with a pre-initialized Stack_head (count = 0 and top = NULL). You will be implementing the functions to push and pop Nodes from the front of the linked list.

Working with Variables

When you get a variable as part of an **assignment**, you will just need to pass its variable field into the **hash_put** function, along with the other token's **val** field. When you get a variable as part of an **operation** (like **+**), you will need to pass it's variable field into the **hash_get** function to get its current value from the symbol table. In this specific case, you'll get back a **Symbol** struct with the current variable and val information. Once you get the **val** from the Symbol, you should use **symbol_free** function from symbol.c to free it, since it's not used anymore.

Hash Table Implementation

You will be implementing your hash table using Separate Chaining with a singly linked list. This table has a header structure that you will be using to access it. This is the **Symtab** struct in symbol.h.

size	The number of Symbols currently in the hash table.
capacity	The number of Indexes allocated for the hash table. (indexes in the array)
table	A pointer to an array of **Symbol *** types. It will be an array of linked lists.

The **Symbol** struct is what your linked list nodes will be made out of. These are also in symbol.h

variable	The name of the variable for this Symbol in the hash table.
val	The value of that variable.
next	A standard next pointer for a singly linked list.

Your function starts out with a pre-initialized Symtab with all table[i] = NULL. Use the provided **hash_code** function to get the hash code, then take that **modulus** the **capacity** to get the index. You will be inserting new Symbols to the end of the linked list at the given index. If the load of the table (size / capacity) is ever >= 2, you will need to rehash into a double-sized symbol table. One of your functions to implement is a rehash function. Since we never remove a symbol, there is no need to reduce the size of the table, only double it if the load is ever >= 2. Since no symbol is ever removed, you will also be returning copies of each symbol once found in the table. The **symbol_copy** function in symbol.c will be very helpful for this.

Recommendations

Step 1: You can get a working calculator with only the code in **rpn.c** and **stack.c** as long as you do not use any variables in your program. The sample program **sample1.txt** does not have any variables. Start with your design for the Stack and the logic for the parse_token function. Implement your logic for TYPE_VALUE, TYPE_OPERATOR, and TYPE_PRINT first. With these and your stack code, you can get a calculator to run for **sample1.txt**. The full, expected output of sample1.txt is included at the end of this document.

Step 2: Once you have all of this code running well for values, operators, and print, then implement your hash table functions and add in support for TYPE_VARIABLE and TYPE_ASSIGNMENT. You should be able to get **sample2.txt** to run once you add in variable support. The full, expected output of sample2.txt is also included at the end of this document.

Step 3: Once all of that is running well, use the tools you know for debugging and make sure your code is running well for all cases, including rehashing. You can write your own sample programs as long as you are careful when writing your postfix-notation equations.

Step 4: If everything is running smoothly and you know your logic is good, the last step is to run your program with **valgrind** to check for memory leaks. You should see this last line if everything was freed.

```
kandrea@zeus-2:testing$ valgrind ./calc sample2.txt
==14257== Memcheck, a memory error detector
....
==14257== All heap blocks were freed -- no leaks are possible
```

Functions to Implement

You will be editing and submitting only three files, **rpn.c**, **stack.c**, and **hash.c**. You may add any helper functions you like to these files, but you cannot change the Makefile or any header files. Below is a quick summary of the functions you will need to complete:

rpn.c:

static int read_file(char *filename, char *line)

- This function should open the file for reading, read the line into **line** (up to MAX_LINE_LEN) and then close the file.
- Return 0 if everything is good.
- On any memory or file errors, return -1.

static int parse_token(Symtab *symtab, Stack_head *stack, Token *tok)

- This is your main function to parse each token in the file you just read.
  - Our pre-written code in **rpn()** gets each token and calls this function to parse them.
- This function receives a pre-initialized symtab and stack and the next Token **tok** to parse.
- Implement the **rules** (on page 2) based on the token type and perform the operations.
- *It is recommended to break this up into many helper functions.*
- Notes:
  - When you use **stack_push**, you have to pass in the stack and tok.
  - When you use **hash_put**, you have to pass in symtab, the variable name, and value.
  - For **hash_get**, you will only pass in the variable name, but you will get a **Symbol** back.
  - When you no longer need a Token, free it using **token_free**.
  - When you no longer need a Symbol, free it using **symbol_free**.
- For TYPE_PRINT, use the **print_step_output(val)** function to print it.
  - This function is used to help generate the output to match this documentation.

stack.c:

Stack_head *stack_initialize()

- This function should malloc a new **Stack_head** and initialize it:
  - Set its count to 0 and next pointer to NULL.
- Return the new Stack_head pointer or NULL on any memory errors.

void stack_destroy(Stack_head *head)

- This function should free an entire Stack (nodes and head).
- When freeing Nodes, remember to call **token_free** on each token.

int stack_push(Stack_head *stack, Token *tok)

- This function pushes a Token on to your stack.
- Create a new Node using **node_create** and passing in **tok** to it.
- Insert this new Node at the **top** of your stack.
- Remember to adjust the counter.
- Return 0 if everything worked fine, or -1 on any errors.

`Token *stack_pop(Stack_head *stack)`

- This function pops and returns a Token from your stack.
- Remove the top Node from the stack and return a pointer to the Token.
- Free the Node using **node_free**. (This will not free the Token).
- Remember to adjust the counter.
- Return the Token pointer if everything worked fine, or NULL on any errors.

`Token *stack_peek(Stack_head *stack)`

- This function returns the top Token pointer on your stack.
- Return the Token pointer if everything worked fine, or NULL on any errors.

`int stack_is_empty(Stack_head *stack)`

- This function returns 1 if the stack is empty or 0 otherwise.

hash.c:

`Symtab *hash_initialize()`

- This function should malloc a new **Symtab** and initialize it:
    - Allocate space for a new table in your symtab.
        - Allocate space for this table on the Heap dynamically.
        - This should be an array to hold HASH_TABLE_INITIAL indexes.
        - The array type is **Symbol ****, meaning each index holds a Symbol Pointer.
        - This will be an array of Linked Lists!
        - Initialize all index elements to NULL.
    - Set its capacity to HASH_TABLE_INITIAL and its size to 0.
- Return the new Symtab pointer or NULL on any memory errors.

`void hash_destroy(Symtab *symtab)`

- This function should destroy an entire **Symtab**:
    - Make sure to go through each index of **table** and free any linked list Symbols in there.
- Free the table and the symtab.

`int hash_get_capacity(Symtab *symtab)`

- This function should return the current number of indexes total in your table.

`int hash_get_size(Symtab *symtab)`

- This function should return the current number of Symbols in your table.

`int hash_put(Symtab *symtab, char *var, int val)`

- This will either add a new variable to the hash table or update an exsiting one.
- Make sure to use **hash_code** and modulus to get the index when hashing.
    - The **hash_code** function is pre-written for you and will turn a variable name into a long.
    - You get the **index** by getting the hash code and then taking it modulus the **capacity**.
- If you already have this variable in the hash table, update its val to the new **val**.
- Otherwise, check to see if the load is >= 2.0
    - **load** is **size / capacity**. Make sure to do this operation as floats or doubles!

- If load >= 2.0, you need to double the capacity and rehash.
    - You will be writing a hash_rehash function to perform this.
- At this point, create the new Symbol using **symbol_create** and insert it at the index.
    - When inserting a new Symbol, add it to the **end** of the Linked List at the index.
- Return 0 if everything went fine, or -1 on any errors.

`Symbol *hash_get(Symtab *symtab, char *var)`

- This will return a **copy** of the Symbol found in the Hash Table (or NULL if not found).
- Make sure to use **hash_code** and modulus to get the index when hashing.
    - The **hash_code** function is pre-written for you and will turn a variable name into a long.
    - You get the **index** by getting the hash code and then taking it modulus the **capacity**.
- Check the linked list at the index for a Symbol with a matching **var**.
    - If found, use the **symbol_copy** function to copy and return a pointer to the copy.
    - If not found, return NULL.

`void hash_rehash(Symtab *symtab, int new_capacity)`

- This will double the capacity in your symtab's table.
    - The passed in **new_capacity** is the new capacity. So, call this with 2* the old capacity.
- You **cannot** use realloc here. This has to be a new table and re-hashing of each Symbol into it.
    - You have to create a new table with new_capacity of Symbol *
    - Then initialize all of the values inside of this new table to NULL
    - Then walk through the old table and put each Symbol's data into the new table.
    - Make sure to free all of the old Symbols and the old Table.
    - Update the symtab's table to point to the new table.

General:

For each function, if you get passed in a Symtab, Stack, Node, or Token that you need, always check if it is NULL first. If it is NULL and shouldn't be, then return as specified in the function description using the code for any errors (usually return -1 or NULL). If the function is void, you can just return.

There are a lot of malloced structs in play when running this code. The most important thing to do is to read the code and understand the context that your functions are called from and what they should be doing with respect to the data structures. Don't worry about freeing unneeded Tokens or Symbols at the beginning. Make sure you have a good program before you put time into making sure you are cleaning everything up.

Also, start with no-variable inputs. Just implement the **rpn.c** and **stack.c** functions and steps needed for values, operators, and print. Once you have your output that looks like the output (at the end of this document) for sample1.txt, then do your implementation for **hash.c** and add the functionality into rpn.c for handling variables and working with your hash table.

Implementation Notes

- **You will be working with Linked Lists and a combination of single and double-pointers.** Make sure to take time to think about how you want to work with the linked list for each function that you need to implement. Make sure you handle all of the cases that could occur.

Submitting and Grading

Submit your **rpn.c**, **hash.c**, and **stack.c** on **blackboard** as a TAR file called **calc.tar**.

TAR creation instructions: **`tar -cvf calc.tar rpn.c stack.c hash.c`**
Example:

```
kandrea@zeus-1:testing$ tar -cvf calc.tar rpn.c stack.c hash.c
rpn.c
stack.c
hash.c
kandrea@zeus-1:testing$ ls calc.tar
calc.tar
```

Be very careful when using tar. Do not change any of the orders or you may accidentally erase one of your files. Always back up your files before doing anything for the first time. (And back them up as you are working on them as well!)

No other naming formats are accepted, it has to be **calc.tar**. Be sure this tar file contains everything you need -- **incomplete submissions cannot be graded**.

Make sure to put your name and G# as a commented line in the beginning of your rpn.c, stack.c, and hash.c files. Also, in the beginning of your program list the known problems with your implementation in a commented section.

All submissions will be **compiled**, **tested, and graded on Zeus!** Make sure you test your code on Zeus before submitting. If your program does not compile on zeus, we cannot grade it. If your program compiles but does not run or generate output on zeus, we cannot grade it.

Questions about the specification should be directed to the CS 531 Piazza forum, which is the primary location for discussions on this project. However, recall that debugging your program is essentially your responsibility; so please do not post long code segments to Piazza. **Do not post any code on Piazza in a public post.** Always post as a **private** post to **Instructors** for code questions. Any general questions about the assignment can be posted publicly.

You **have one late token** that may be used on any project in the course. No late submissions are allowed without using a token, and you only have one token for all three projects. This is meant for personal emergencies, so plan on submitting on time and saving your token for any emergencies throughout the semester.

Your grade will be determined as follows:

- **80 points** - Correctness. This is graded by automated script by checking the results of each of your functions as you complete them. You will get partial credit for most of the functions, so even if something is not working at the end, you may still get credit for any working code.

- **20 points** - Code & comments: Be sure to document your design clearly in your code comments. This score will be based on (subjective) reading of your source code by the GTA. The grader will also evaluate your C programming style according to the below guidelines.

Test your program by running **calc** with different input programs (eg. **./calc sample2.txt**) and looking at the output. Make sure it is selecting the right process for each time stamp. Feel free to add print statements (or use the **gdb** debugger) to make sure you are performing each of the functions properly. It is your responsibility to make sure to test and check your functions.

If your program does not compile, it will get a very low grade (just the code & comments points).

Code Style Guidelines (Will be Part of your Code and Comments Score)

1. No Global Variables may be used. (Global Constants are allowed; eg. const int val = 100;)
2. Always initialize all pointers with a value or to NULL on declaration.
3. Each block of code should increase the indent by 2-4 spaces.
4. Only use one statement per line. (eg. int x = 42; int y = 32; would be wrong)
5. Use Braces { } around all if/if else/else statements, even if they only have one statement.
6. Always check the return value for a call to malloc() to make sure it is not NULL
7. Set pointers to NULL after freeing them.
8. ***(Recommendation only)*** Functions **should** be fairly short (20 lines of code). If you have large functions that perform several operations, you should consider breaking them into smaller functions.

sample1.txt Contents

1 2 + 3 4 - 5 6 + 4 2 / * * + print

Expected Output:

```
######### Beginning Program (sample1.txt) ###########

.-------------------
| Program Step =  0
|-----Program Remaining
| 1 2 + 3 4 - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  1
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 1
|-----Program Remaining
| 2 + 3 4 - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  2
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 2 1
|-----Program Remaining
| + 3 4 - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  3
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 3
|-----Program Remaining
| 3 4 - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  4
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 3 3
|-----Program Remaining
| 4 - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  5
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 4 3 3
|-----Program Remaining
| - 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  6
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| -1 3
|-----Program Remaining
| 5 6 + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  7
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 5 -1 3
|-----Program Remaining
| 6 + 4 2 / * * + print
```

```
o-------------------

.-------------------
| Program Step =  8
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 6 5 -1 3
|-----Program Remaining
| + 4 2 / * * + print
o-------------------

.-------------------
| Program Step =  9
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 11 -1 3
|-----Program Remaining
| 4 2 / * * + print
o-------------------

.-------------------
| Program Step = 10
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 4 11 -1 3
|-----Program Remaining
| 2 / * * + print
o-------------------

.-------------------
| Program Step = 11
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 2 4 11 -1 3
|-----Program Remaining
| / * * + print
o-------------------

.-------------------
| Program Step = 12
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 2 11 -1 3
|-----Program Remaining
| * * + print
o-------------------

.-------------------
| Program Step = 13
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 22 -1 3
|-----Program Remaining
| * + print
o-------------------

.-------------------
| Program Step = 14
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| -22 3
|-----Program Remaining
| + print
o-------------------

.-------------------
| Program Step = 15
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| -19
|-----Program Remaining
| print
o-------------------
```

```
.-------------------
| Program Step = 16
|-----Program Output
| -19
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
|
|-----Program Remaining
o-------------------
```

sample2.txt Contents

x 3 2 + = foo 4 = x foo + print

Expected Output:

```
######### Beginning Program (sample2.txt) ###########

.-------------------
| Program Step =  0
|-----Program Remaining
| x 3 2 + = foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  1
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| x
|-----Program Remaining
| 3 2 + = foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  2
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 3 x
|-----Program Remaining
| 2 + = foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  3
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 2 3 x
|-----Program Remaining
| + = foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  4
|-----Symbol Table [0 size/5 cap]
|-----Program Stack
| 5 x
|-----Program Remaining
| = foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  5
|-----Symbol Table [1 size/5 cap]
|          x: 5
|-----Program Stack
|
|-----Program Remaining
| foo 4 = x foo + print
o-------------------

.-------------------
| Program Step =  6
|-----Symbol Table [1 size/5 cap]
|          x: 5
```

```
|-----Program Stack
| foo
|-----Program Remaining
| 4 = x foo + print
o-------------------

.-------------------
| Program Step =  7
|-----Symbol Table [1 size/5 cap]
|          x: 5
|-----Program Stack
| 4 foo
|-----Program Remaining
| = x foo + print
o-------------------

.-------------------
| Program Step =  8
|-----Symbol Table [2 size/5 cap]
|          x: 5
|        foo: 4
|-----Program Stack
|
|-----Program Remaining
| x foo + print
o-------------------

.-------------------
| Program Step =  9
|-----Symbol Table [2 size/5 cap]
|          x: 5
|        foo: 4
|-----Program Stack
| x
|-----Program Remaining
| foo + print
o-------------------

.-------------------
| Program Step = 10
|-----Symbol Table [2 size/5 cap]
|          x: 5
|        foo: 4
|-----Program Stack
| foo x
|-----Program Remaining
| + print
o-------------------

.-------------------
| Program Step = 11
|-----Symbol Table [2 size/5 cap]
|          x: 5
|        foo: 4
|-----Program Stack
| 9
|-----Program Remaining
| print
o-------------------

.-------------------
| Program Step = 12
|-----Program Output
| 9
|-----Symbol Table [2 size/5 cap]
|          x: 5
|        foo: 4
|-----Program Stack
|
|-----Program Remaining
o-------------------
```