CS 2110 Homework 5
Intro to Assembly

Chris, Sam, Soha, Prindle, and Henry

Spring 2020

Contents

1 Overview

1.1 Purpose:

So far in this class, you have seen how binary or machine code manipulates our circuits to achieve a goal. However, as you have probably figured out, binary can be hard for us to read and debug, so we need an easier way of telling our computers what to do. This is where assembly comes in. Assembly language is symbolic machine code, meaning that we don't have to write all of the ones and zeros in a program, but rather symbols that translate to ones and zeros. These symbols are translated with something called the assembler. Each assembler is dependent upon the computer architecture on which it was built, so there are many different assembly languages out there. Assembly was widely used before most higher-level languages and is still used today in some cases for direct hardware manipulation.

1.2 Task:

The goal of this assignment is to introduce you to programming in LC-3 assembly code. This will involve writing small programs, translating conditionals and loops into assembly, modifying memory, manipulating strings, and converting high-level programs into assembly code.

You will be required to complete the four functions listed below with more in-depth instructions on the following pages:

1. `mult.asm`
2. `selectionsort.asm`
3. `printvowels.asm`
4. `linkedlist.asm`

1.3 Criteria:

Your assignment will be graded based on your ability to correctly translate the given pseudocode into LC-3 assembly code. Check the deliverables section for deadlines and other related information. Please use the LC-3 instruction set when writing these programs. More detailed information on each instruction can be found in the Patt/Patel book Appendix A (also on Canvas under LC3 Resources). Please check the rest of this document for some advice on debugging your assembly code, as well some general tips for successfully writing assembly code.

You must obtain the correct values for each function. While we will give partial credit where we can, your code must assemble with **no warnings or errors** (Complx will tell you if there are any). If your code does not assemble, we will not be able to grade that file and you will not receive any points. Each function is in a separate file, so you will not lose all points if one function does not assemble. Good luck and have fun!

2 Detailed Instructions

2.1 Part 1: Implementing Multiply

To start you off with this homework, we are implementing the multiply function! Store the result of the operation in the label `ANSWER`. Arguments A and B are stored in memory, and you will load them from there to perform this operation. Assume the values of A and B are positive integers. Implement your assembly code in `mult.asm`.

Suggested Pseudocode:

```
a = (argument 1);
b = (argument 2);
ANSWER = 0;
while (b > 0) {
    ANSWER = ANSWER + a;
    b--;
}
// note: when the while-loop ends, the value stored at ANSWER is a times b.
```

2.2 Part 2: Selection Sort

The second assembly function is to selection-sort all elements of an array in memory. Use the pseudocode to help plan out your assembly and make sure you are sorting it properly! Implement your assembly code in `selectionsort.asm`. For more information on selection sort and a good visual representation, check out this link.

Suggested Pseudocode:

```
x = 0;                          // current swapping index in the array
len = length of array;
while(x < len - 1) {
    z = x;                      // index of minimum value in unsorted portion of array
    y = x + 1;                  // current index in array
    while (y < len) {
        if (arr[y] < arr[z]) {
            z = y;              // update the index of the minimum value
        }
        y++;
    }
    if (z != x) {
        temp = arr[x];          // perform a swap
        arr[x] = arr[z];
        arr[z] = temp;
    }
    x++;
}
```

2.3 Part 3: Printing Vowels

The third assembly function is to print the vowels in a **null-terminated** string. The label `STRING` will contain the **address** of the first character of the string. Remember that strings are just arrays of consecutive characters. Implement your assembly code in `printvowels.asm`

Assume every character in the string is UPPERCASE.

To check for these vowel characters, **refer to the ASCII table** and remember that each of these characters are represented by a word (16-bits) in the LC-3's memory. This is a **null-terminated** string, meaning that a 0 will be stored immediately after the final character in memory!

NOTE:

0 is the same as '\0'
0 is different from '0'

Suggested Pseudocode:

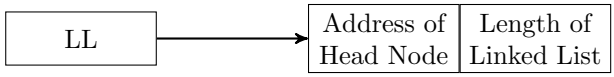
```
string = "TWENTY ONE TEN";
i = 0;
while(string[i] != '\0'){
    if(string[i] == 'A' || string[i] == 'E' ||
        string[i] == 'I' || string[i] == 'O' ||
        string[i] == 'U'){

        print(string[i]);
    }
    i++;
}
```

Hint: Take a look at the trap vectors in Appendix A for printing characters to the console.

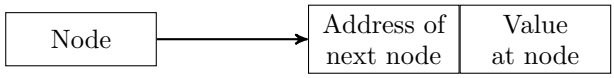
2.4 Part 4: Make elements of a Linked List into a string

For the final problem, your goal is to create a string from the elements of a linked list and store that string at the label named `ANSWER` (don't forget about the null terminating character mentioned in Part 3). In order to do so, look at the label we have given you:

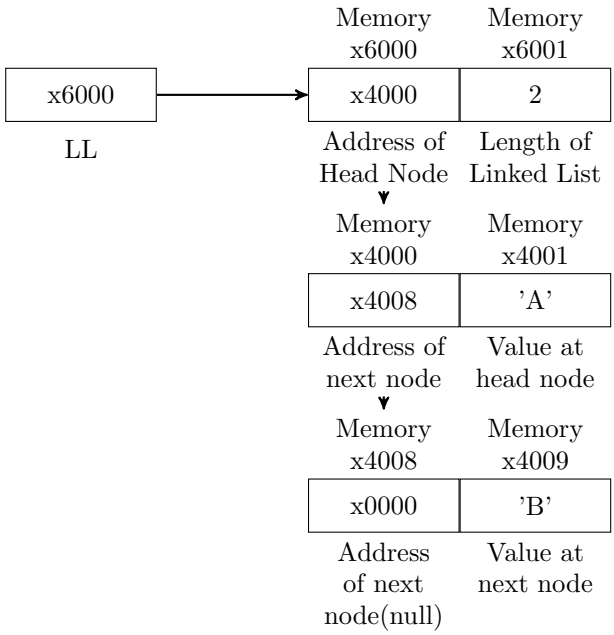
- LL: The address of a Linked List object. Similar to Java, our linked list is an object with two attributes - head and length. These two attributes are stored in memory like so:



  Every node in our linked list is another object with two attributes - next node and value. These two attributes are stored in memory like so:



  So together, our data structure would look something like this:



Now that you understand what data structure we are dealing with, we have provided the following pseudocode to help you begin your coding! This code will be implemented in the `linkedlist.asm` file.

Suggested Pseudocode:

```
length = LL.length;
curr = LL.head;                 //HINT: What can an LDI instruction be used for?
ANSWER = char[length];          //This character array will already be setup for you
i = 0;
while (curr != null) {
    ANSWER[i] = curr.value;
    curr = curr.next;
    i++;
}
```

3 Deliverables

Turn in the files

1. `mult.asm`
2. `selectionsort.asm`
3. `printvowels.asm`
4. `linkedlist.asm`

on Gradescope by February 25th at 11:55pm.

There will be a 6 hour late period until February 26th at 5:55am. You will receive a 25% deduction during this late period. Anything after that will result in an automatic 0.

Note: Please do not wait until the last minute to run/test your homework, history has proved that last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned.

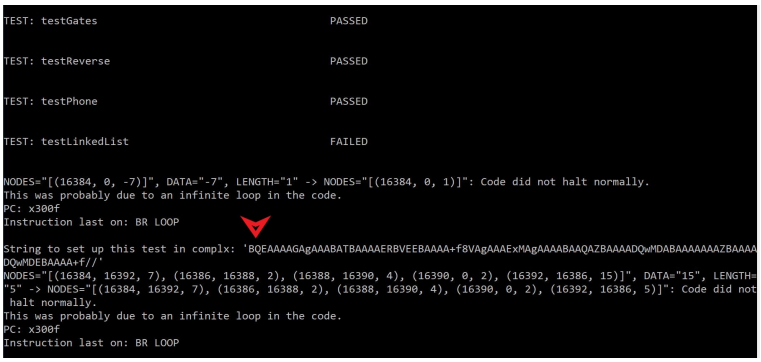
4 Running Autograder and Debugging LC-3 Assembly

When you turn in your files on gradescope for the first time, you might not receive a perfect score. Does this mean you change one line and spam gradescope until you get a 100? No! You can use a handy tool known as tester strings.

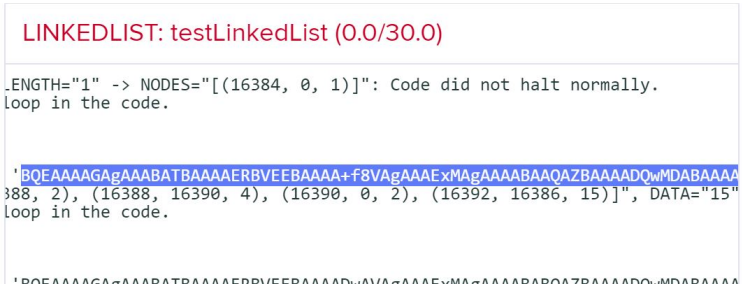
1. First off, we can get these tester strings in two places: the local grader or off of gradescope. To run the local grader:

   - Mac/Linux Users:
     (a) Navigate to the directory your homework is in. **In your terminal, not in your browser**
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) On **docker quickstart**, navigate to the directory your homework is in
     (b) Run `./grade.sh`

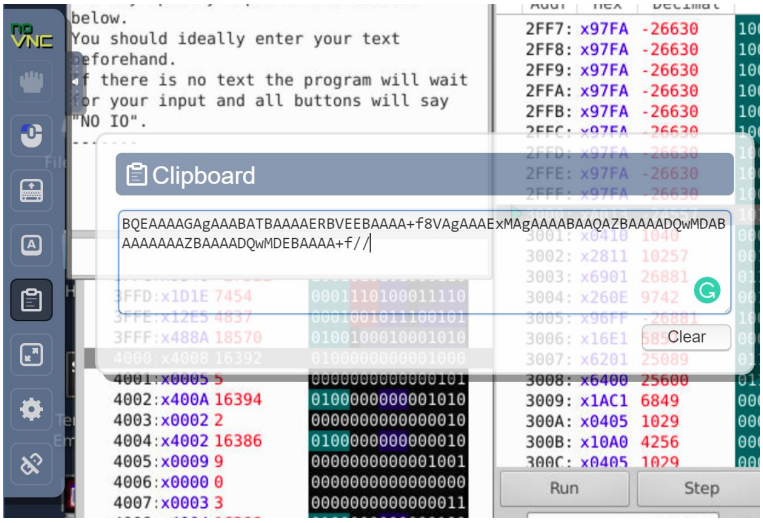
   When you run the script, you should see an output like this:



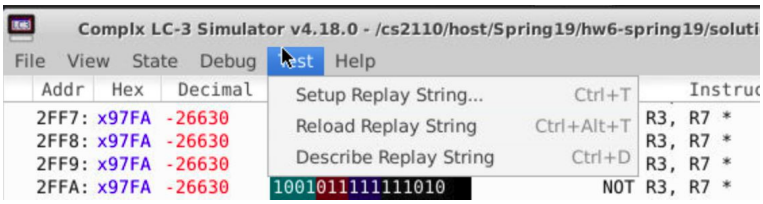
   Copy the string, starting with the leading 'B' and ending with the final backslash. Do not include the quotation marks.
   Side Note: If you do not have docker installed, you can still use the tester strings to debug your assembly code. In your gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslace and again, don't copy the quotations.



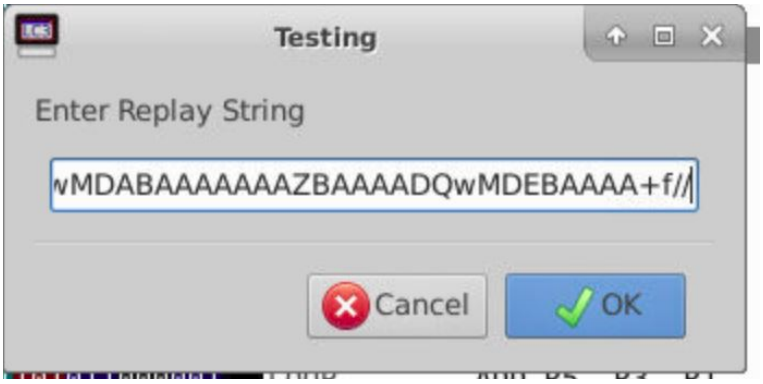
2. Secondly, navigate to the clipboard in your docker image and paste in the string.



3. Next, go to the Test Tab and click Setup Replay String

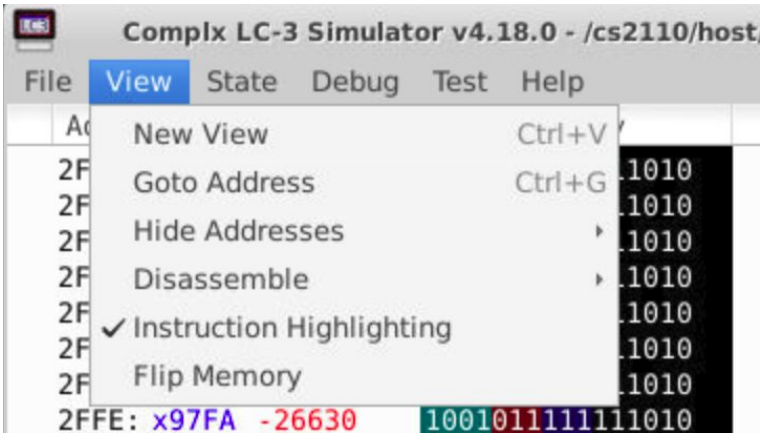


4. Now, paste your tester string in the box!

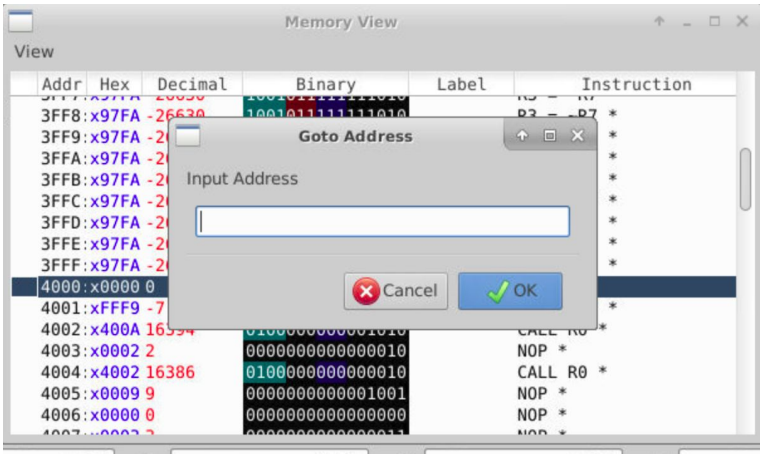


5. Now, complx is set up with the test that you failed! The nicest part of complx is the ability to step through each instruction and see how they change register values. To do so, click the step button. To change the number representation of the registers, double click inside the register box.

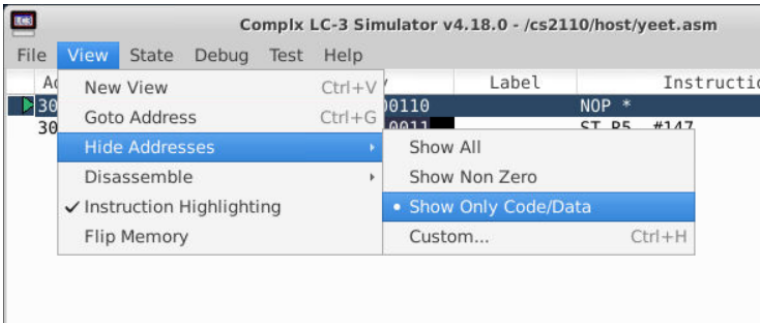
6. If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'



7. Now in your new view, go to the area of memory where your data is stored by CTRL+G and insert the address



8. One final tip: to automatically shrink your view down to only those parts of memory that you care about (instructions and data), you can use View Tab → Hide Addresses → Show Only Code/Data. Just be careful: if you misclick and select Show Non Zero, it *may* make the window freeze (it's a known Complx bug).



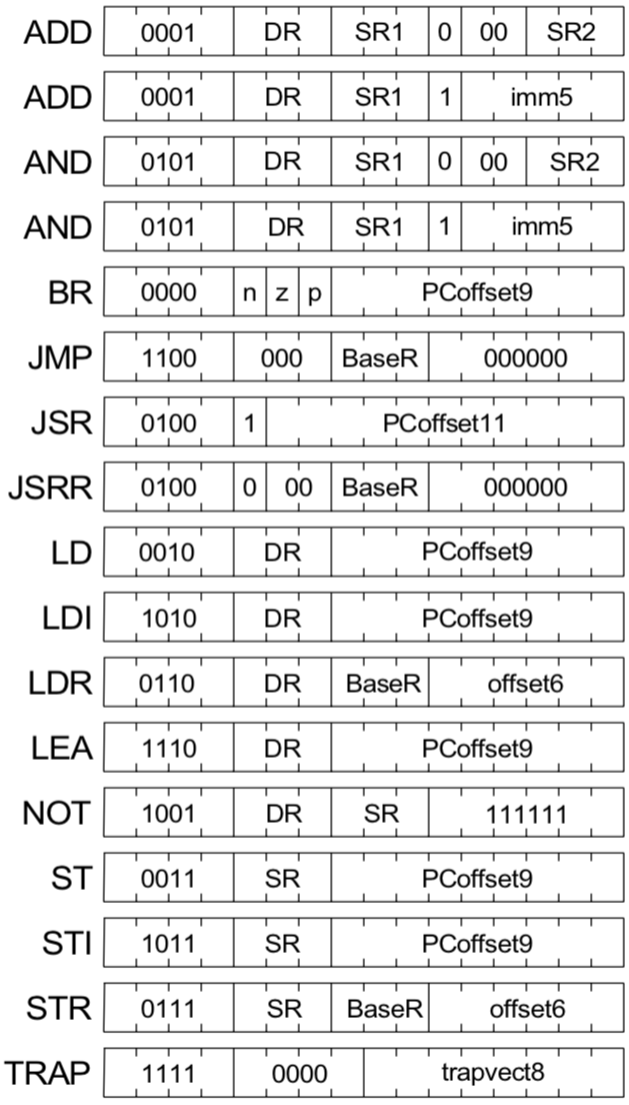
5 Appendix

5.1 Appendix A: ASCII Table

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

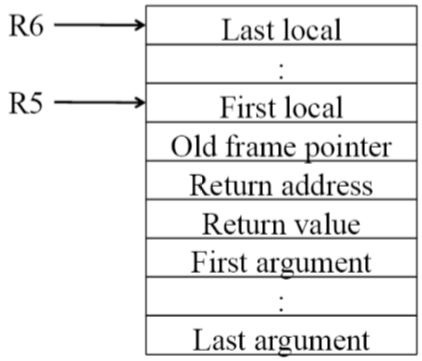
Figure 1: ASCII Table — Very Cool and Useful!

5.2 Appendix B: LC-3 Instruction Set Architecture



Trap Vector	*Assembler Name*
x20	GETC
x21	OUT
x22	PUTS
x23	IN
x25	HALT

Device Register	*Address*
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06



5.3 Appendix C: LC-3 Assembly Programming Requirements and Tips

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will complain if there are any issues. **If your code does not assemble you WILL get a zero for that file.**

2. **Comment your code!** This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad you left yourself notes on what certain instructions are contributing to the code. Comment things like what registers are being used for and what less intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semicolon (;), and the rest of that line will be a comment.

3. Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

   Good Comment

   ```
   ADD R3, R3, -1          ; counter--
   BRp LOOP                ; if counter == 0 don't loop again
   ```

   Bad Comment

   ```
   ADD R3, R3, -1          ; Decrement R3
   BRp LOOP                ; Branch to LOOP if positive
   ```

4. **DO NOT assume that ANYTHING in the LC-3 is already zero.** Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.

5. Following from 3. You can randomize the memory and load your program by doing File - Randomize and Load.

6. Use the LC-3 calling convention. This means that all local variables, frame pointer, etc... must be pushed onto the stack. Our autograder will be checking for correct stack setup.

7. Start the stack at xF000. **The stack pointer always points to the last used stack location.** This means you will allocate space **first**, then store onto the stack pointer.

8. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).

9. Do not add any comments beginning with @plugin or change any comments of this kind.

10. **Test your assembly.** Don't just assume it works and turn it in.

5.4 Appendix D: Rules and Regulations

5.4.1 General Rules

1. Starting with the assembly homeworks, any code you write should be meaningfully commented for your benefit. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

5.4.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.

3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

4. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

5.4.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas/Gradescope, nor

your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

5.4.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

5.4.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.

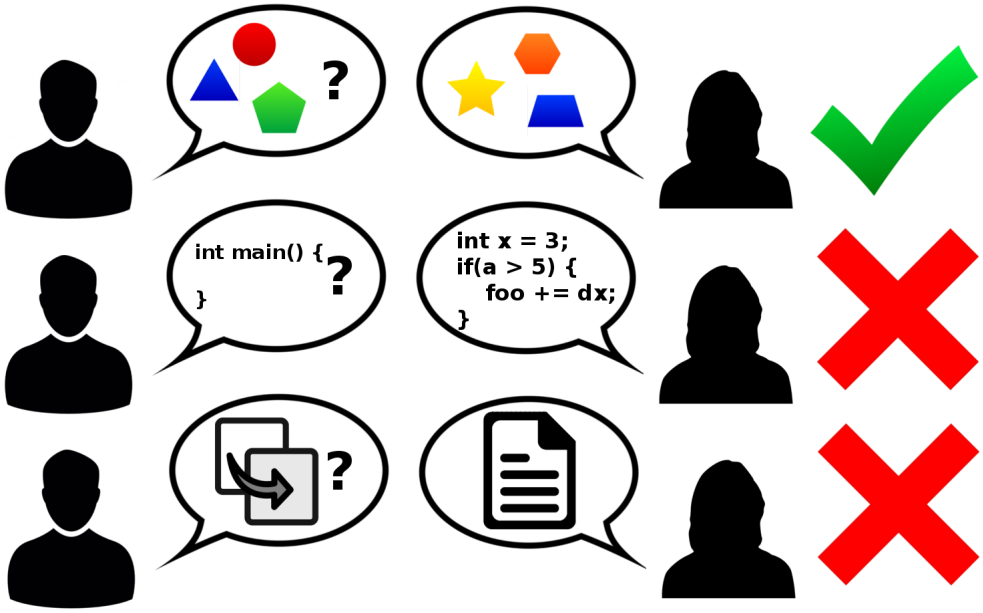


Figure 2: Collaboration rules, explained colorfully