CS 2110 Homework 6
Subroutines and Calling Conventions

Prabhav Gupta, Lydia Lazor, Jonathan Marto,
Richard So, Jeff Shelton, Srishti Grover, Shivani Vora

Spring 2024

Contents

1 Overview

1.1 Purpose

Now that you've been introduced to assembly, think back to some high level languages you know such as Python or Java. When writing code in Python or Java, you typically use functions or methods. Functions and methods are called subroutines in assembly language.

In assembly language, how do we handle jumping around to different parts of memory to execute code from functions or methods? How do we remember where in memory the current function was called from (where to return to)? How do we pass arguments to the subroutine, and then pass the return value back to the caller?

The goal of this assignment is to introduce you to the Stack and the Calling Convention in LC-3 Assembly. This will be accomplished by writing your own subroutines, calling subroutines, and even creating subroutines that call themselves (recursion). By the end of this assignment, you should have a strong understanding of the LC-3 Calling Convention and the Stack Frame, and how subroutines are implemented in assembly language.

1.2 Task

You will implement each of the three subroutines (functions) listed below in LC-3 assembly language. Please see the detailed instructions for each subroutine on the following pages. The autograder checks for certain subroutine calls with arguments pushed in the correct order, so we suggest that you follow the provided algorithms when writing your assembly code. Your subroutines must adhere to the LC-3 calling convention.

1. `GCD.asm`
2. `isPalindrome.asm`
3. `DFS.asm`

Later in this document, you can also find some general tips for successfully writing assembly code.

1.3 Criteria

Your assignment will be graded based on your ability to correctly translate the given pseudocode for subroutines (functions) into LC-3 assembly code, following the LC-3 calling convention. Please use the LC-3 instruction set when writing these programs. Check the deliverables section for deadlines and other related information.

You must obtain the correct values for each function. In addition, registers R0-R5 and R7 must be restored from the perspective of the caller, so they contain the same values before and after the caller's JSR call. Your subroutine must return to the correct point in the caller's code, and the caller must find the return value on the stack where it is expected to be. If you follow the LC-3 calling convention correctly, each of these things will happen automatically.

Your code must assemble with no warnings or errors. (LC3Tools 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. Good luck and have fun!

2 Detailed Instructions

2.1 Part 1

2.1.1 GCD

The GCD of a pair of numbers is the greatest number that is a divisor for both numbers. In `GCD.asm`, we want you to implement two subroutines: `MOD` and `GCD` (see the following pseudocodes for more details). Note that the numbers that we will give you are strictly greater than or equal to 0. As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

Remember to get your arguments from the stack!

For example:
GCD(12, 32) should return 4
GCD(50, 20) should return 10
GCD(2, 1) should return 1
GCD(0, 1) should return 1

2.1.2 Pseudocode

Here are the pseudocodes for these subroutines:

```
MOD(int a, int b) {
    while (a >= b) {
        a -= b;
    }
    return a;
}

GCD(int a, int b) {
    if (b == 0) {
        return a;
    }

    while (b != 0) {
        int temp = b;
        b = MOD(a, b);
        a = temp;
    }
    return a;
}
```

Note: Since there are two loops in the `MOD` and `GCD` subroutines, make sure you use different label names for those loops. In general, you should not have two labels with the same name in the same file.

2.2 Part 2

2.2.1 isPalindrome

For this part of this assignment, you will be checking whether an array of chars is a palindrome in `isPalindrome.asm`. As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

You may be wondering . . . didn't you already do something like this in HW5? Yes, but this time, **you must do it recursively**.

2.2.2 Pseudocode

Here is the pseudocode for this subroutine:

```
Parameters:
    - addr string: the starting address of the string
    - len:  the length of the string (not including the null terminator)

Returns: 1 if the string is a palindrome, 0 otherwise.

IS_PALINDROME(addr string, len) {
    if (len == 0 || len == 1) {
        return 1;
    } else {
        if (string[0] == string[len - 1]) {
            return IS_PALINDROME(string + 1, len - 2);
        } else {
            return 0;
        }
    }
}
```

Lets do an example to see how our array looks throughout our above algorithm!

isPalindrome(['r','o', 't', 'o', 'r'])

$\boxed{\text{'r'}}\boxed{\text{'o'}}\boxed{\text{'t'}}\boxed{\text{'o'}}\boxed{\text{'r'}}$ → isPalindrome(['o', 't', 'o'])
$\boxed{\text{'o'}}\boxed{\text{'t'}}\boxed{\text{'o'}}$ → isPalindrome(['t'])
$\boxed{\text{'t'}}$ → Return 1

2.3 Part 3

2.3.1 DFS

For this part of the assignment, you will write a recursive subroutine for a graph in `DFS.asm`. The parameters of the function are the graph's root node (provided as an address in memory), and the target node we are searching for (the node's data, **not** the address). As a reminder, please do **not** change the names of provided subroutines or otherwise your submission will not pass the autograder.

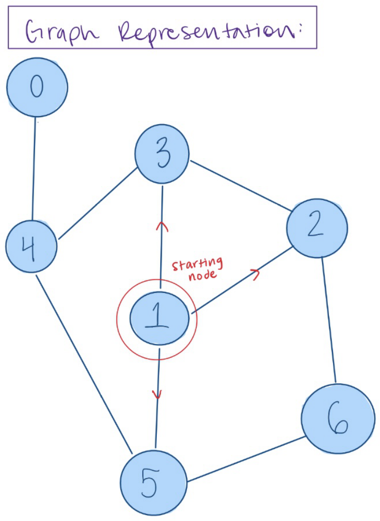
If you are unfamiliar with DFS and would like more information about it, see https://www.baeldung.com/cs/depth-first-search-intro.

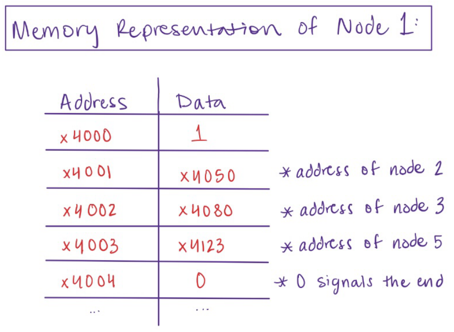
2.3.2 Graph Data Structure

The below figures show us the visual representation of a graph, as well as the memory representation for a node. We are able to implement this abstract graph data structure as a collection of nodes - where each node holds the following information: its data (aka its number/value), and the addresses of its neighboring nodes (nodes that it is connected to via a line or an edge). Thus, each node is really a memory address, representing the start of the chunk of memory where it will store this information.

In this example, we can look at the starting node (1) in our graph. At memory address x4000, it stores the node's data (= 1), at address x4001 it will store the address of its neighboring node (2), at address x4002 the address of its neighboring node (3), and at address x4001 the address of its neighboring node (5).

Note: We will know that we have gone through all of the neighboring nodes if the next address has a value of 0.




Memory Representation of Node 1:

Address	Data	
x4000	1	
x4001	x4050	* address of node 2
x4002	x4080	* address of node 3
x4003	x4123	* address of node 5
x4004	0	* 0 signals the end
...	...	

2.3.3 Visited Set and Visited Vector

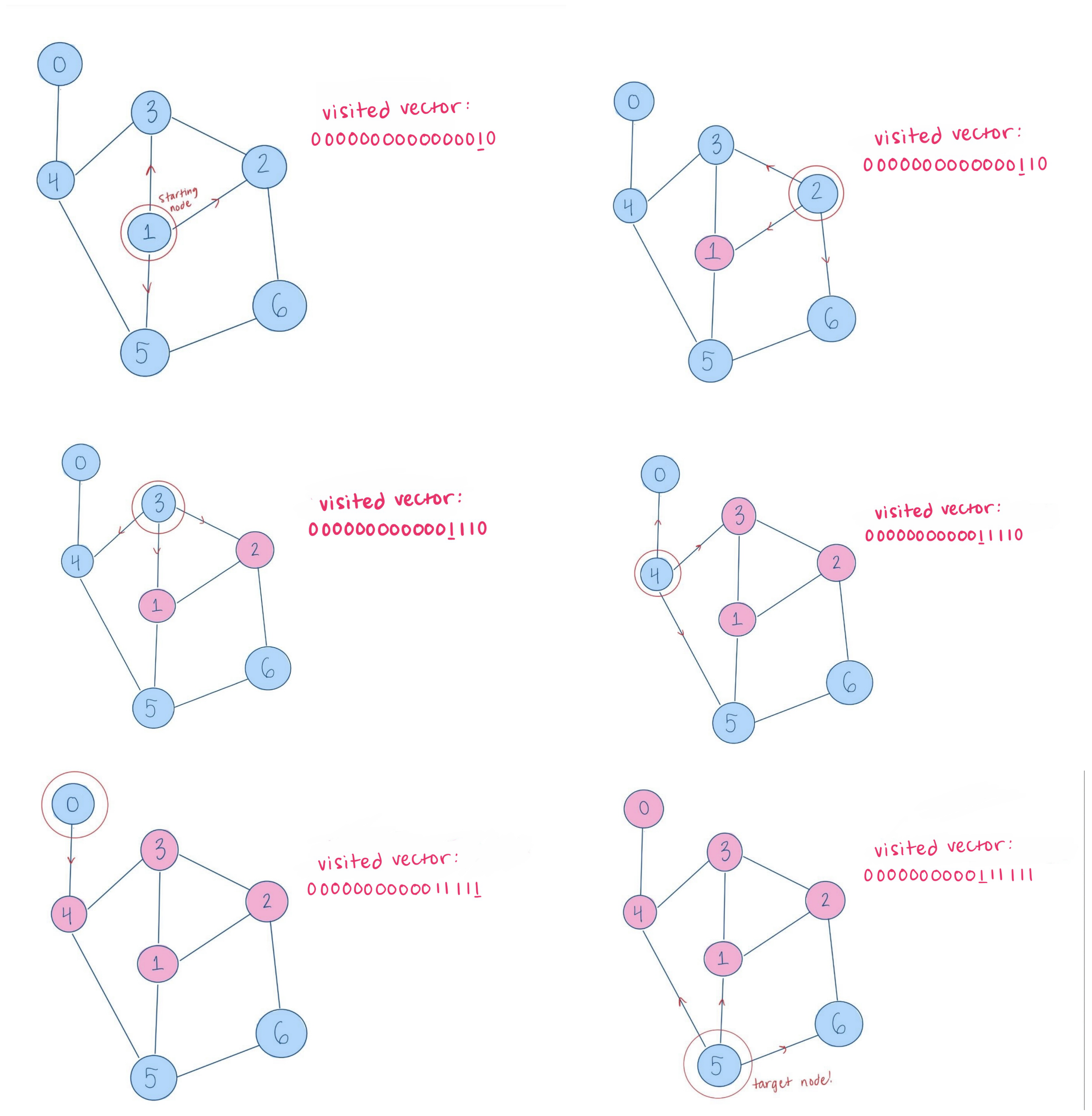
In the context of graph algorithms, we often make use of a visited set to keep track of all the nodes that have been visited. This allows us to efficiently check in O(1) time whether a node has already been traversed.

Considering the constraints of LC-3 assembly and the overhead required to implement such a data structure, we will instead be using a **visited vector**. This vector is a 16-bit binary number that is going to be stored at a specific address in memory, so that it will persist outside of the scope of a single subroutine. Within the vector, each bit corresponds to a node, i.e. bit 0 corresponds to node 0, bit 5 corresponds to node 5, etc. If a bit is 0, that node is not part of the visited set, and if it is 1, that means the node has already been visited. This leads to the obvious constraint that the visited vector can only represent up to 16 nodes. **All test cases will use 16 nodes or less to ensure that a single vector is sufficient**.

In order to manage the visited vector, you will first implement 2 helper functions: IS_VISITED() and SET_VISITED(). The pseudocode for these functions can be found below.

Let's see an example of DFS in action on this graph. Our starting node will be 1, and our target node is 5.

DFS(Address of Node 1, 5)



2.3.4 Pseudocode

Here is the pseudocode for these subroutines:

```
SET_VISITED(addr node) {
    visited = mem[mem[VISITED_VECTOR_ADDR]];
    data = mem[node];
    mask = 1;
    while (data > 0) {
        mask = mask + mask;
        data--;
    }
    mem[mem[VISITED_VECTOR_ADDR]] = (visited | mask); //Hint: Use DeMorgan's Law!
}

IS_VISITED(addr node) {
     visited = mem[mem[VISITED_VECTOR_ADDR]];
     data = mem[node];
     mask = 1;
     while (data > 0) {
         mask = mask + mask;
         data--;
     }
     return (visited & mask) != 0;
}

DFS(addr node, int target) {
      SET_VISITED(node);
      if (mem[node] == target) {
         return node;
      }
      result = 0;
      for (i = node + 1; mem[i] != 0 && result == 0; i++) {
          if (! IS_VISITED(mem[i])) {
              result = DFS(mem[i], target);
          }
      }
      return result;
}
```

Note: Since there are multiple loops and conditionals in the all of the subroutines in this file, make sure you use different label names for those loops. In general, you should not have two labels with the same name in the same file.

3 Autograder

For this homework, there are a variety of test cases associated with each assembly file you implement. Additionally, there are two ways to grade your submission: locally, or through Gradescope.

To run the local grader:

1. Ensure that you have Docker running.
2. Navigate to the directory your homework is in.
3. If you are on MacOS or Linux, run the command `sudo chmod +x grade.sh`
4. Now run `./grade.sh` if you are on MacOS/Linux, or `.\grade.bat` on Windows.

When you turn in your files on Gradescope for the first time, you may not receive a perfect score. Does this mean you change one line and spam Gradescope until you get a 100? No!

Each test case details exactly how we are testing your assembly code (e.g. writing to labels, writing strings, expecting a certain answer, etc.). Here is an example:

```
Writing word "touppercase" at MEM[MEM[WORD]]
Writing start argument '2' at MEM[START]
Writing end argument '9' at MEM[END]
Running student assembly...

--Outputs correct answer: Expected: toUPPERCAse, Got: touppercase
```

You can use this information to replicate such tests yourself locally on LC3Tools.

Note: The checker may not reflect your final grade on this assignment. We reserve the right to update the autograder as we see fit when grading.

4 Deliverables

Turn in the following files to Gradescope:

1. `GCD.asm`
2. `isPalindrome.asm`
3. `DFS.asm`

Make sure that you are submitting .asm files!! The autograder will not grade .obj files.

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

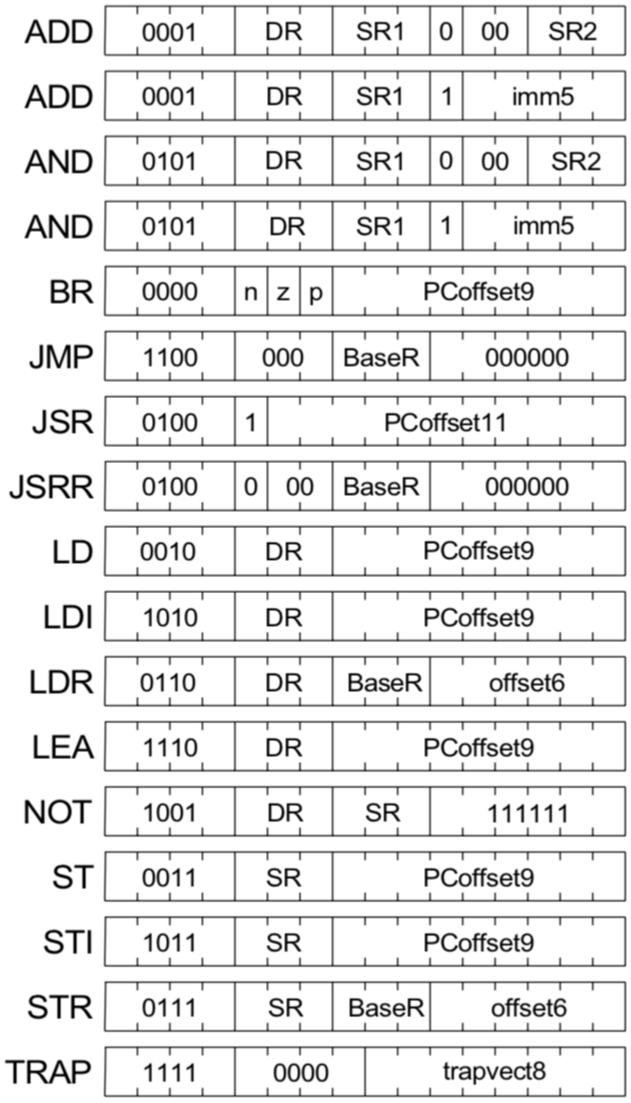
5 Demos

This homework will be demoed. The demos will be 10 minutes long and will occur **IN PERSON**. Stay tuned for details as the due date approaches.

Please note: Your grade for this assignment is split into 2 parts. 50% of your grade is based upon how well you do with the autograder. The other 50% is determined by how well you do in the demo. Again, more details to come soon.

6 Appendix

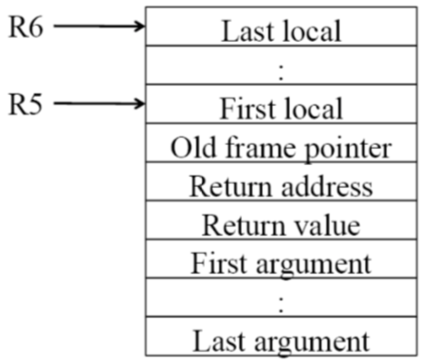
6.1 Appendix A: LC-3 Instruction Set Architecture



ADD	0001	DR	SR1	0	00	SR2
ADD	0001	DR	SR1	1	imm5	
AND	0101	DR	SR1	0	00	SR2
AND	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD	0010	DR	PCoffset9			
LDI	1010	DR	PCoffset9			
LDR	0110	DR	BaseR	offset6		
LEA	1110	DR	PCoffset9			
NOT	1001	DR	SR	111111		
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			

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



6.2 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 LC3Tools. 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.
3. **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. The autograder will do the same.
4. As you step through your assembled code in LC3Tools, registers or memory locations that have updated after one step will be highlighted to you. Use this to your advantage to figure out bugs in your assembly if you are failing a test case.
5. Do NOT execute any data as if it were an instruction (meaning you should put `.fills` after `HALT` or `RET`). All your program does is interpret the values RAM as instructions until it reaches HALT. If you use .fill before your program HALTS, your program might interpret what you filled as an instruction and try to execute it!
6. **Test your assembly.** Don't just assume it works and turn it in.
7. When translating pseudocode into assembly, don't skip over the closing brackets! Even though they're only one character long, perhaps they also might need to be translated into assembly...

7 Rules and Regulations

1. Please read the assignment in its entirety before asking questions.
2. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
3. If you find any problems with the assignment, please report them to the TA team. Announcements will be posted if the assignment changes.
4. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency please reach out to your instructor and the head TAs **IN ADVANCE** of the due date with documentation (i.e. note from the dean, doctor's note, etc).
5. You are responsible for ensuring that what you turned in is what you meant to turn in. 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 Gradescope. Email submissions will not be accepted.
6. See the syllabus for information regarding late submissions; any penalties associated with unexcused late submissions are non-negotiable.

7.1 Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work. Homework assignments will be examined using cheat detection programs to find evidence of unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student. If you supply a 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 platform which would allow other parties to it (public repositories, pastebin, etc). If you would like to use version control, use a private repository on github.gatech.edu

Homework collaboration is limited to high-level 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.

High-level collaboration means 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, or providing other students any part of your code.

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.

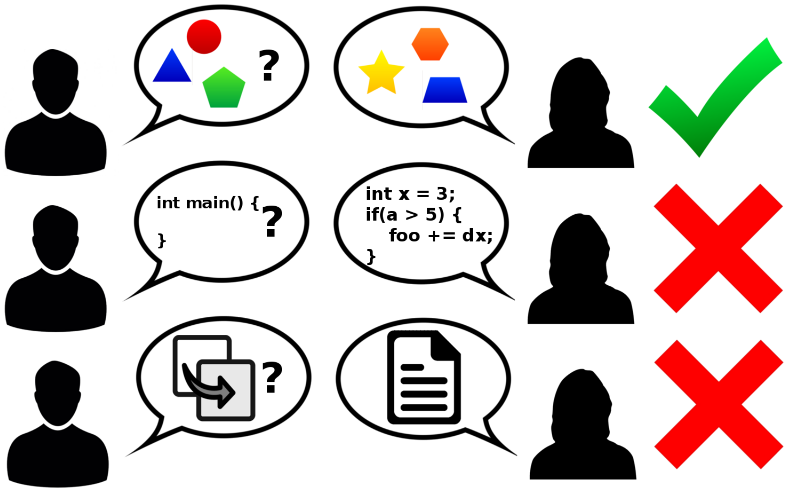


Figure 1: Collaboration rules, explained colorfully