Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. Concurrent programming helps reduce bugs by organizing code into independent threads of execution. **T** **(F)**
2. On a uniprocessor, it is safe to use Rc<T> instead of Arc<T>. **T** **(F)**
3. In Java, the synchronized keyword enables communicating between threads. **T** **(F)**
4. The Rust compiler can automatically detect and prevent deadlocks from occurring. **T** **(F)**
5. Trait objects can only be created on the heap. **(T)** **F**
6. Rust has language support for concurrent programming. **T** **(F)**

Question 2. Multiple Choices

Pick all answer(s) that are correct.

a) Which of the following functions are examples of a `fold` function?

i. `reverse`

(ii.) `sum`

(iii.) `any`

iv. `map`

v. `zip`

Question 3. Programming Questions

1. In a peculiar home with mice and cats, they share an extra-large bowl of food together. The rules at this house is that the cats must eat alone, while the mice can eat together, up to 6 of them at once. When a cat wants to eat, it has priority over any mouse that wants to eat. The mice who are already eating must finish up and leave. If another cat is eating, the other cat must wait for its turn.

   Given the following shared data structure:

```
enum Eater { NONE, CATS, MICE, }
struct Bowl {
    eater: Eater, // which kind of animal is eating right now
    count: i32,   // number of animals eating together (mice only)
    ncats: i32,   // number of cats waiting
}
```

a) Given the following code snippet:

```
1  struct Monitor {
2      mutex: Mutex,
3      cv: Condvar,   // waiting for the bowl
4  }
5
6  const NCATS = 2;
7
8  fn main() {
9      let mut threads = vec![];
10     let bowl = Bowl { eater: NONE, count: 0, ncats: 0 };
11     let monitor = Rc::new(Monitor {
12         mutex: Mutex::new(bowl),
13         cv: Condvar::new(),
14     });
15
16     for i in 1..=NCATS {
17         threads.push(thread::spawn(|| {
18             for _ in 0..10 {
19                 cat_eat(i, &monitor);
20             }
21         }));
22     }
23
24     /* create mouse threads here */
25
26     for child in threads {
27          child.join().unwrap();
28     }
29 }
```

The above code currently does not compile. Identify the line(s) where error(s) would occur by specifying the line number(s), and if possible, write the code that would fix it.

Line #	Corrected Line
1	`Monitor<T>`
2	`Mutex<T>`
6	`const NCATS: i32 = 2;`
11	`Arc::new`
17	`let monitor = monitor.clone();`
17	`thread::spawn(move || { … });`

b) Complete the `cat_eat` function by filling in the blanks.

```
fn cat_eat(i: u64, monitor: _____ &Arc<Monitor<Bowl>> _____) {
    let Monitor {mutex, cv} = &**monitor;
    let mut bowl = mutex.lock().unwrap();

    while __________ bowl.eater != NONE _____________________ {
        bowl.ncats += 1;

        bowl = _____ cv.wait(bowl).unwrap()_________; // wait its turn
        bowl.ncats -= 1;
    }
    bowl.eater = CATS;
    bowl.count = 1;

    _____________ drop(bowl) _________________; // unlock
    cat_eating();   // lock should not be held during this call
    let mut bowl = mutex.lock().unwrap();
    bowl.eater = NONE;
    bowl.count = 0;

    _____________ cv.notify_all() ____________; // wake up everyone
}
```

c) Complete the `mouse_eat` function in the spaces provided below.

```
fn mouse_eat(/* function parameters redacted */)
{
    let Monitor {mutex, cv} = &**monitor;
    let mut bowl = mutex.lock().unwrap();


    loop {
        if bowl.eater != CATS && bowl.ncats == 0 && bowl.count < 6 {
            break;
        }

        bowl = cv.wait(bowl).unwrap();
    };

    bowl.eater = MICE;
    bowl.count += 1;
    drop(bowl);




    mouse_eating(); // lock should not be held during this call
    let mut bowl = mutex.lock().unwrap();




    bowl.count -= 1;
    if bowl.count <= 0 {
        bowl.eater = NONE;
    }
    cv.notify_all();
```

2. Complete the following generic function, `average`, such that it returns the arithmetic average of the type parameter T. Your solution must make use of the `fold` iterator adaptor and must be able to handle an empty list correctly without crashing.

```
fn average<T>(list: & Vec<T>) -> T
where T: Copy + Add<Output=T> + Default + From<u32> + Div<Output=T>
{



    if list.len() == 0 {
        Default::default()
    }
    else {
        list.iter().fold(Default::default(), |ac: T, v: &T| ac + *v)
            / From::from(list.len() as u32)
    }




}
```

3. The Shape trait has one method, area, which returns the area of the shape. Write a function which takes a list of shapes and returns the average area. Then, implement the Shape trait for Rectangle, Circle, and Triangle.

See average.rs

4. The `lower` function takes a string slice, converts it to lower case, removes all non-alphabetic characters, and returns a String object. Complete this function in Rust using only iterators and iterator adaptors without using any control flow statements (e.g. if, else, for, .etc). Hint: `char::to_lowercase()` and `char::is_lowercase()` may be useful.

```
fn lower(word: &str) -> String {



    word.chars().map(|c| c.to_ascii_lowercase())
                .filter(|c| c.is_lowercase())
                .collect()



}
```

5. Use `mpsc::channel` and multiple threads to improve the performance of large square matrix multiplication.

See mmult.rs

6. In the dining philosopher problem, there are *N* philosophers and *N* chopsticks in between each pair of philosophers. As you may know, you need a pair of chopsticks to be able to eat. A philosopher must successfully acquire both chopsticks to his/her left and right before proceeding to eat. Simulate this problem by creating one thread per philosopher, and use a monitor to synchronize the use of chopsticks.

See dining.rs