Sheet 4

1. Write a function `find_equal`, that takes two `&str` as input `s1` and `s2`. The function look for a string slice of length 2 that appear in both `s1` and `s2`. If successful the function returns a tuple with the two equal slices. one from `s1`, the other one from `s2`. otherwise it return `None`

   Then write a second function `lucky_slice`, that take a n `&str` named `input_str` as input. The function create a String with the same length as `input_str`, filled with random lowercase letters, and then call `find_equal` on the two strings. If `find_equal` is successful the function `lucky_slice` return the slice of `input_str` that was found by `find_equal`. otherwise it returns `None`
2. Write a struct `Person` that has 3 fields. a `name` with type `String`, and two parents (`father` and `mother`). Each parent is an `Option` of an immutable reference to a `Person`. THen implement the following methods:

   - a `new` method, that takes a name and two options to the parents and then returns a new `Person`.
   - a `find_relatives` method that take a `u32` input named `generations`. the function returns a `Vec` containing all the relatives within the generations range... for example:
     - if `generations` is 0, the return should be just the person itself
     - if `generations` is 1, the return should be the person itself + his parents
     - if `generations` is 2, the function should return person itself + his parents + his grandparents.
   - a `find_roots` method that returns a `Vec` containing all the relatives that has at least one parent set to `None`

   implement the second and the third methods recursively
3. Correct the following code. You can edit only the indicated line

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a, 'b> ImportantExcerpt<'a> { //THIS LINE
    fn announce_and_return_part(&'a self, announcement: &'b str) -> &'b str
{
        println!("Attention please: {}", announcement);
        self.part
```

```
    }
}
```

4. Annotate struct with lifetime:
   - `r` and `s` must have different lifetimes
   - lifetime of `s` is bigger than that of `r`

```
struct DoubleRef<T> {
    r: &T,
    s: &T
}
```

5. Split Trait
   Write a trait split that has one generic type `ReturnType`. the trait has one method `split` that take
   a immutable reference to `self` and return a tuple `(ReturnType,ReturnType)`. the function split the
   element of the trait in half.
   Implement the trait for:
   - `String`, where `split` returns `(&str, &str)`
   - `&[i32]`, where `split` returns `(&[i32], &[i32])`
   - `LinkedList<f64>`, where `split` returns `(LinkedList<f64>, LinkedList<f64>)`
6. Geometry
   Create the following structs:

   - `Point` that has a `x` and a `y` coordinates
   - `Circle` that has a `radius` and a `center`
   - `Rectangle` that has a `top_left` and a `bottom_right` points
     Implement the trait `Default` (that is already defined in the standard library)
   - A default point is a point center in (0,0)
   - A default circle is has a center in (0,0) and radius one
   - A default rectangle has the points in (-1,1) and (1,-1)

   for `Point` implement the traits `Add` and `Sub` (that are the trait for the `+` and `-` operators).
   For example:

   - (10,11) + (1,1) = (11,12)
   - (0,0) - (20,30) = (-20,-30)

   Create the struct `Area` that contains a f32 value with the area. Then implement the `Default` trait that returns am area with 0

Implement the trait `Display` for `Area` with a custom message looking like: `Area is 20 cm²`

Create a custom trait `GetArea` that adds a method `get_area(&self)->Area` to the 3 objects.
The method returns the geometric area of the shape (a Point has area 0)

implement the `Add` trait that for:

- Summing an `Area` with an `Area`
- Summing an `Area` with an `&dyn GetArea`

create a function `sum_area` that take as input a slice of `&dyn GetArea` object and returns the area
of all objects summed

7. Create a function 'skip_prefix' that, given non mutable references to `telephone_number: &str` and `prefix: &str`, returns `&str`. The function removes the prefix from the number, and if there isn't the prefix contained at the start of `number`, return `number` itself.
8. Create a struct `Chair` that has two fields: `color: &str` and `quantity: &usize`. Create another structure `Wardrobe` that has the same fields of `Chair`. Create a trait `Object` that has two function declarations: `build` that has `&self` as argument and returns a `&str`; `get_quantity` that has `&self` as argument and return a `String`.
   Then, implement the trait `Object` for `Chair` and `Wardrobe`.

   - `build` should return a `&str` with "Chair/Wardobe has been built"
   - `get_quantity` should return a formatted message with the number of chairs/wardrobes.
     Implement also the Display trait for Chair and Wardrobe, that returns a different formatted message if there are zero, one or two or more chairs/wardrobes.
     The message should also contain the color of the objects if there are one or more.

   Pay attention to the lifetimes!
9. Create a simple permission manager for certain actions on an Operating System.
   Create an enum `Role` that has the following fields: `GUEST, USER, ADMIN`. Create another enum called `Permission` with the following fields: `READ, WRITE, EXECUTE`. For these enums derive the trait `PartialEq`, `Eq` and, for Permission, `Hash`. You'll need them later for some comparison and for using the enum Permission as a key of an HashMap.
   Create a struct `Actions` that has the field `action: String` and `permission: HashMap<Permission, bool>`.

   Create then the struct `User` with these fields: `name: String`, `role: Role`, `actions: Vec<Actions>`
   Write the trait `Auth` with the following methods:

- `check_permission` with arguments `&self`, `action: &str`, `permission_type: &Permission` and returns a bool. This method checks if there is an action in self with as a name the string passed in the arguments. If it exists return if the permission_type for that action is true or false. If this action doesn't exist in self, then return false
- `can_write` with arguments `&self` and `string &str` and returns a bool. This method use `check_permission` and checks if an actions identified with the string argument is writeable.
- `can_read` with arguments `&self` and `string &str` and returns a bool. This method use `check_permission` and checks if an actions identified with the string argument is readable.
- `can_execute` with arguments `&self` and `string &str` and returns a bool. This method use `check_permission` and checks if an actions identified with the string argument is executable

Apply the trait `Auth` for `User`, and write the relative methods implementations.

Implement the `Default` trait for Actions. Write the method `default` that return `Self` and set the action field with an empty string and the permission field with an hashmap that contains as keys the Permission values `READ, WRITE, EXECUTE`, and, as values, all set to `false`.

Create the method `new` for Actions, that, given the arguments `action: String`, `read: bool`, `write: bool` and `execute: bool`, return Self. In this method, create a new Self, setting the action field to `action` argument and the `permission` field with the key-value pairs corresponding to the three Permission values (`READ, WRITE, EXECUTE`) and the relative bool passed as arguments.

Implement the `Default` trait for User. Write the method `default` that return `Self` and set the name field with the string "Guest", the Role with GUEST and the actions field as a new vector.

Create also the method `change_role` for `User` that take a mutable reference to self and the argument `role: Role`, and return `Result<(), String>`

This method should

- change the User's role as the one passed in the argument if the user is `ADMIN`, returning an `Ok(())`
- otherwise if the user is `USER` its role can be modified to `GUEST` or it can remain `USER`, returning an `Ok(())`, but cannot be changed to `ADMIN`. In this last case it should return a `Err()` with an error message
- Lastly if the user is `GUEST` its role cannot be changed, can be only re-set to `GUEST`, returning an `Ok(())`. Otherwise, it should return a `Err()` with an error message

Create the function `sudo_change_permission` that takes as arguments `user: User`, `string: String`, `permission: Permission` and `value: bool`.

This function should change the user permission for the action specified in the `string` argument.