Sheet 6 Solutions

1. Create a struct `TreeNode` generic over `T` that represents a binary tree.
   It should have a field `value` of type `T` and two optional fields `left` and `right` (they should hold a pointer to another `TreeNode`).
   Implement:

   - a method `new` that takes a value and returns a new `TreeNode` with the given value and no children.
   - a method `from_vec` that takes a vector of values and returns a `TreeNode` with the given values.
   - a method `insert` that takes a value and inserts it into the tree (follow binary search tree rules).

   Implement the preorder, inorder and postorder traversal algorithms for the tree.

   Keep in mind that the type `T` must implement the `PartialOrd` and `Clone` trait_es.

```rust
use std::fmt::Debug;

#[derive(Debug)]
struct TreeNode<T: PartialOrd + Clone + Debug> {
    value: T,
    left: Option<Box<TreeNode<T>>>,
    right: Option<Box<TreeNode<T>>>,
}

impl<T> TreeNode<T>
where
    T: PartialOrd + Clone + Debug,
{
    pub fn new(value: T) -> Self {
        TreeNode {
            value,
            left: None,
            right: None,
        }
    }

    pub fn from_vec(vec: &[T]) -> Self {
        let mut tree = TreeNode::new(vec[0].clone());
        for value in vec.iter().skip(1) {
            tree.insert(value.clone());
        }
```

```rust
        tree
    }

    pub fn insert(&mut self, value: T) {
        if value < self.value {
            match self.left {
                Some(ref mut left) => left.insert(value),
                None => self.left = Some(Box::new(TreeNode::new(value))),
            }
        } else {
            match self.right {
                Some(ref mut right) => right.insert(value),
                None => self.right = Some(Box::new(TreeNode::new(value))),
            }
        }
    }

    pub fn preorder(&self) {
        println!("{:?}", self.value);
        if let Some(ref left) = self.left {
            left.preorder();
        }
        if let Some(ref right) = self.right {
            right.preorder();
        }
    }

    pub fn inorder(&self) {
        if let Some(ref left) = self.left {
            left.inorder();
        }
        println!("{:?}", self.value);
        if let Some(ref right) = self.right {
            right.inorder();
        }
    }

    pub fn postorder(&self) {
        if let Some(ref left) = self.left {
            left.postorder();
        }
        if let Some(ref right) = self.right {
            right.postorder();
        }
        println!("{:?}", self.value);
    }
}

#[cfg(test)]
```

```rust
mod tree_tests {
    use super::*;

    #[test]
    fn normal_tree() {
        let mut tree = TreeNode::new(4);
        tree.insert(2);
        tree.insert(5);

        println!("{:?}", tree);
    }

    #[test]
    fn tree_from_vec() {
        let vec = vec!['d', 'c', 'b', 'a', 'e', 'g', 'f'];
        let tree = TreeNode::from_vec(&vec);

        println!("{:?}", tree);
    }

    #[test]
    fn tree_preorder() {
        let vec = vec!['d', 'c', 'b', 'a', 'e', 'g', 'f'];
        let tree = TreeNode::from_vec(&vec);

        tree.preorder();
    }

    #[test]
    fn tree_inorder() {
        let vec = vec![10, 3, 5, 2, 1, 4, 6, 7];
        let tree = TreeNode::from_vec(&vec);

        tree.inorder();
    }

    #[test]
    fn tree_postorder() {
        let vec = vec!['d', 'c', 'b', 'a', 'e', 'g', 'f'];
        let tree = TreeNode::from_vec(&vec);

        tree.postorder();
    }
}
```

2. Create a struct Car with the following fields:

- `model: String`,
- `year: u32`,
- `price: u32`,
- `rent: bool`

Create a struct CarDealer with a field that is a vector of `Car`.
Create a struct `User` with a field that is an `Option` of `Car`.
Implement the following methods for `CarDealer`:

- `new` that takes a vector of Car and returns a CarDealer
- `add_car` that takes a Car and adds it to the vector of Car
- `print_cars` that prints all the cars
- `rent_user` that takes a mutable reference to a User and a model: String, that identify the car, and assigns the car to the user and set the rent field to true. If the car is not found, print "Car not found".
  The car **must be** the same present in the vector of CarDealer and into the car field of the User.
- `end_rental` that takes a mutable reference to a User and set the rent field to false. If the user has no car, print "User has no car".

Implement the `new` and `default` method for Car
Implement the `print_car` method for User that prints the car if it is present, otherwise print "User has no car"

```rust
use std::cell::RefCell;
use std::rc::Rc;

type CarRef = Rc<RefCell<Car>>;

#[derive(Debug)]
struct Car {
    model: String,
    year: u32,
    price: u32,
    rent: bool,
}

impl Car {
    pub fn new(model: String, year: u32, price: u32, rent: bool) -> Self {
        Self {
            model,
            year,
            price,
            rent,
        }
```

```rust
    }
    pub fn default() -> Self {
        Self {
            model: "".to_string(),
            year: 0,
            price: 0,
            rent: false,
        }
    }
}

struct CarDealer {
    cars: Vec<CarRef>,
}

struct User {
    car: Option<CarRef>,
}

impl CarDealer {
    pub fn new(cars: Vec<CarRef>) -> Self {
        Self { cars }
    }

    pub fn add_car(&mut self, car: Car) {
        self.cars.push(Rc::new(RefCell::new(car)))
    }

    pub fn print_cars(&mut self) {
        self.cars.iter_mut().for_each(|x| {
            println!("{:?}", x);
        })
    }

    pub fn rent_user(&mut self, user: &mut User, model: String) {
        let mut index = 0;
        let mut found = false;

        for (i, car) in self.cars.iter().enumerate() {
            if car.borrow_mut().model == model {
                index = i;
                if !found {
                    found = true;
                }
            }
        }

        if found {
            let clone_car: CarRef = Rc::clone(&self.cars[index].clone());
```

```
            println!("index: {:?}", index);
            println!("clone_car: {:?}", clone_car);

            let a = clone_car.clone();
            clone_car.borrow_mut().rent = true;

            user.car = Some(a);
        } else {
            println!("Car not found");
            return;
        }
    }

    pub fn end_rental(&mut self, user: &mut User) {
        match user.car.clone() {
            Some(car) => {
                car.borrow_mut().rent = false;
                user.car = None;
            }
            None => {
                println!("User has no car");
                return;
            }
        }
    }
}

impl User {
    pub fn print_car(&self) {
        match self.car.clone() {
            Some(car) => {
                println!("{:?}", car.borrow());
            }
            None => {
                println!("User has no car");
                return;
            }
        }
    }
}

#[test]
fn test_car_dealer() {
    //create cars
    let car1 = Car {
        model: "Audi".to_string(),
        year: 2010,
        price: 10000,
        rent: false,
```

```rust
    };
    let car2 = Car {
        model: "BMW".to_string(),
        year: 2015,
        price: 20000,
        rent: false,
    };
    let car3 = Car {
        model: "Mercedes".to_string(),
        year: 2018,
        price: 30000,
        rent: false,
    };

    let mut car_dealer = CarDealer::new(vec![
        Rc::new(RefCell::new(car1)),
        Rc::new(RefCell::new(car2)),
        Rc::new(RefCell::new(car3)),
    ]);

    let mut user = User { car: None };

    car_dealer.print_cars();

    car_dealer.rent_user(&mut user, "BMW".to_string());

    user.print_car();

    assert_eq!(car_dealer.cars[1].borrow_mut().rent, true);

    car_dealer.print_cars();

    car_dealer.end_rental(&mut user);

    car_dealer.print_cars();

    assert_eq!(car_dealer.cars[0].borrow_mut().rent, false);
}
```

3. Write the trait_es `Sound` that defines a method `make_sound` that returns a `String`.
   Create some structs that implement the `Sound` trait_es (animals).
   Create a list of trait_es objects that implement the `Sound` trait_es via the struct `FarmCell`.
   The struct `FarmCell` should have a field `element` containing the trait_es object and a field `next` that holds an optional pointer to another `FarmCell`.
   Implement the methods:

- `new` for the struct `FarmCell` that takes a trait_es object and returns a new `FarmCell`.
- `insert` for the struct `FarmCell` that takes a trait_es object and inserts it into the list (push_back).

Implement the trait_es `Sound` for the struct `FarmCell` that returns the concatenation of the `make_sound` methods of all the elements in the list.

```
trait Sound {
    fn make_sound(&self) -> String;
}

struct Dog;
struct Cat;
struct Frog;
struct Cow;

impl Sound for Dog {
    fn make_sound(&self) -> String {
        format!("woof woof")
    }
}

impl Sound for Cat {
    fn make_sound(&self) -> String {
        format!("meow meow")
    }
}

impl Sound for Frog {
    fn make_sound(&self) -> String {
        format!("croak croak")
    }
}

impl Sound for Cow {
    fn make_sound(&self) -> String {
        format!("moo moo")
    }
}

struct FarmCell {
    element: Box<dyn Sound>,
    next: Option<Box<FarmCell>>,
}

impl FarmCell {
```

```rust
    pub fn new(element: Box<dyn Sound>) -> Self {
        FarmCell {
            element,
            next: None,
        }
    }

    pub fn insert(&mut self, element: Box<dyn Sound>) {
        match self.next {
            Some(ref mut next) => next.insert(element),
            None => self.next = Some(Box::new(FarmCell::new(element))),
        }
    }
}

impl Sound for FarmCell {
    fn make_sound(&self) -> String {
        let mut result = self.element.make_sound();
        if let Some(ref next) = self.next {
            result.push_str(&format!(" {}", next.make_sound()));
        }
        result
    }
}

#[cfg(test)]
mod sound_list_tests {
    use super::*;
    #[test]
    fn test_list() {
        let mut list = FarmCell::new(Box::new(Dog));
        list.insert(Box::new(Cat));
        list.insert(Box::new(Frog));
        list.insert(Box::new(Cow));
        // println!("{}", list.make_sound());

        assert_eq!(list.make_sound(), "woof woof meow meow croak croak moo moo");
    }
}
```

4. create the struct `PublicStreetlight` with the fields `id: &str`, `on: bool` and `burn_out: bool`: it represent a public light, with its id, if it is on or off and if it is burned out or not. Create the struct `PublicIllumination` with the field `lights` that is a vector of `PublicStreetlight`.

   Implement the methods `new` and `default` for `PublicStreetlight` and `PublicIllumination`. Then implement the Iterator trait for `PublicIllumination` that returns the burned out lights in order to permit the public operators to change them. The iterator must remove the burned out lights from the vector.

```rust
#[derive(Copy, Clone, Debug, PartialEq)]
struct PublicStreetlight<'a> {
    id: &'a str,
    on: bool,
    burn_out: bool,
}

impl<'a> PublicStreetlight<'a> {
    pub fn new(id: &'a str, on: bool, burn_out: bool) -> Self {
        Self { id, on, burn_out }
    }

    pub fn default() -> Self {
        Self::new("", false, false)
    }
}

struct PublicIllumination<'a> {
    lights: Vec<PublicStreetlight<'a>>,
}

impl<'a> PublicIllumination<'a> {
    fn new(p0: Vec<PublicStreetlight<'a>>) -> Self {
        Self { lights: p0 }
    }
    fn default() -> Self {
        Self { lights: vec![] }
    }
}

impl<'a> Iterator for PublicIllumination<'a> {
    type Item = PublicStreetlight<'a>;

    fn next(&mut self) -> Option<Self::Item> {
        let a = self
            .lights
            .iter()
            .enumerate()
            .find(|&x| x.1.burn_out == true);
        match a {
            Some((i, _)) => {
                let b = self.lights[i];
                self.lights.remove(i);
                Some(b)
            }
```

```
            None => None,
        }
    }
}

#[test]
fn test_1() {
    //create new streetlights
    let streetlight = PublicStreetlight::new("1", true, true);
    let streetlight2 = PublicStreetlight::new("2", true, false);
    let streetlight3 = PublicStreetlight::new("3", true, false);
    let streetlight4 = PublicStreetlight::new("4", false, true);

    let publicIllumination =
        PublicIllumination::new(vec![streetlight, streetlight2, streetlight3,
streetlight4]);

    for a in publicIllumination {
        println!("{:?}", a);
    }
}
```

5. Using the code below as a reference, create a "compile time tree" implementation. you need to:
   - Add the trait bounds
   - implement CompileTimeNode for Node and NullNode
   - implement the function count_nodes that counts the (non_null) nodes of a specific tree type

```
use std::marker::PhantomData;

trait CompileTimeNode{
    type LeftType;
    type RightType;
    fn is_none() -> bool;
}
struct NullNode{}

struct Node<L,R>{
    left: PhantomData<L>,
    right: PhantomData<R>
}

fn count_nodes<T>() -> usize{
    todo!()
}
```

```rust
use std::marker::PhantomData;

trait CompileTimeNode{
    type LeftType: CompileTimeNode;
    type RightType: CompileTimeNode;
    fn is_none() -> bool;
}
struct NullNode{}
impl CompileTimeNode for NullNode{
    type LeftType = NullNode;
    type RightType = NullNode;
    fn is_none() -> bool {
        true
    }
}

struct Node<L: CompileTimeNode,R: CompileTimeNode>{
    left: PhantomData<L>,
    right: PhantomData<R>
}
impl<L: CompileTimeNode,R: CompileTimeNode> CompileTimeNode for Node<L,R>{
    type LeftType = L;
    type RightType = R;
    fn is_none() -> bool{ false }
}

fn count_nodes<T: CompileTimeNode>() -> usize{
    let mut count = 0;
    if !T::is_none(){
        count = 1;
        count += count_nodes::<T::LeftType>();
        count += count_nodes::<T::RightType>();
    }
    count
}

#[test]
fn test(){

    let len = count_nodes::<

        Node<
            Node<
                Node<
                    NullNode,
                    NullNode,
                >,
```

```
                NullNode
            >,
            Node<
                Node<
                    Node<
                        Node<
                            NullNode,
                            NullNode
                        >,
                        NullNode
                    >,
                    Node<
                        NullNode,
                        NullNode
                    >
                >,
                NullNode
            >
        >
    >();
    assert_eq!(len,8)
}
```

6. Create a struct named `EntangledBit`.
Wen two bits `b1` and `b2` are entangled with each-other they are connected, meanings that they will always have the same value.
A bit can be entangled with any number of other bits (including 0)
implement the following functionalities:
    - implement the Default trait for `EngangledBit` that return a bit set to 0, entangled with 0 other bits.
    - implement the methods `set` (set the bit to 1) `reset` (set the bit to 0) and `get` (return true or
false) to manipulate a bit.
    - implement a method `entangle_with(&self, other: &mut Self)` that entangle `other` to `self`.
        - if `other` is entangled with other bits it gets "un-entangled".
        - `other`'s value gets overwritten by the value of `self`

```
use std::cell::RefCell;
use std::rc::Rc;

struct EntangledBit{
    bit: Rc<RefCell<bool>>
}
```

```rust
impl Default for EntangledBit {
    fn default() -> Self {
        Self{
            bit: Rc::new(RefCell::new(false))
        }
    }
}

impl EntangledBit{
    pub fn get(&self) -> bool{
        *self.bit.borrow()
    }
    pub fn set(&mut self){
        *self.bit.borrow_mut() = true;
    }
    pub fn reset(&mut self){
        *self.bit.borrow_mut() = false;
    }
    pub fn entangle_with(&self, other: &mut Self){
        other.bit = self.bit.clone();
    }
}

#[test]
fn test(){
    let mut b1 = EntangledBit::default();
    let mut b2 = EntangledBit::default();
    assert_eq!(b2.get(),false);
    b1.entangle_with(&mut b2);
    b1.set();
    assert_eq!(b2.get(),true);
}
```