Question 2: Refer to the function cons (https://docs.rs/im/5.0.0/im/list/fn.cons.ht and

a- provide an explanation of the function, the answer should provide a description of what the function does, and a detailed explanation of each parameter of the function. ### Answer: The cons function simply prepends a value to a list, adds the value to the front of the list. The first argument it takes is the value, `car` of type `T` to be prepended and the second argument is a list containing elements of type `T`, `cdr: List<T>`, which must be borrowable , of which the `car` is to be added to the front of then it returns a new `List<T>` with `car` in front of the other elements.

Question 3: Rust has a lot of smart pointers, such as Rc, Arc, Cell, and RefCell...

Answer:

The Error: The exact error when executing the original code is:

```
error[E0594]: cannot assign to `task.id`, as `task` is not declared as mutable
  --> src\main.rs:17:5
   |
17 |     task.id=100;
   |     ^^^^^^^^^^^ cannot assign
   |
help: consider changing this to be mutable
   |
13 |     let mut task = Task {
   |         +++
```

By default in Rust, all variables are immutable, so when you create `let task = Task { ... }`, you cannot modify any of its fields unless the variable itself is declared as mutable with `let mut task = ...`.

How Interior Mutability Solves This: Interior mutability is a design pattern in Rust that allows you to mutate data even when there are immutable references to it. Normally, Rust's borrowing rules enforce immutability at compile time, but with interior mutability types like `Cell<T>`, the mutation happens at runtime.

`Cell<T>` provides the following: - It allows you to get and set values internally - It works only with types that are `Copy` (since it copies values in and out) - It wraps the value and provides `get()` to retrieve a copy of the value and `set()` to modify it

Rewritten Code:

```
use std::cell::Cell;
```

```
enum Level {
    Low,
    Medium,
    High
}

struct Task {
    id: Cell<u8>,
    level: Level
}

fn main() {
    let task = Task {
        id: Cell::new(10),
        level: Level::High
    };
    task.id.set(100);
    println!("Task with ID: {}", task.id.get());
}
```

This code works because: 1. We use `Cell<u8>` for the `id` field instead of plain `u8` 2. We initialize `id` with `Cell::new(10)` 3. We modify the value using `task.id.set(100)` instead of direct assignment 4. We read the value using `task.id.get()` which returns a copy

Question 4: Consider the following program:

Answer:

a- What the program is doing

The program creates two nodes (`a` and `b`) using `Rc<RefCell<Option<DoubleNode>>>`. First, `b.next` points to `a`. Then it mutably borrows `a` and sets `a.prev` to point to `b`. This creates a two-way link between the nodes (a cycle). It also prints `Rc` strong reference counts before and after the link is created.

b- What `DoubleNode` is trying to implement

`DoubleNode` is trying to implement a node in a **doubly linked list**: - `value`: stores the node data - `next`: pointer to the next node - `prev`: pointer to the previous node

`Option<DoubleNode>` is used so a pointer can be empty (`None`) at the ends of the list, `Rc` allows shared ownership, and `RefCell` allows mutation at runtime even when shared through `Rc`.

c- Difference between `Weak<RefCell<Option<DoubleNode>>>` and `Rc<RefCell<Option<DoubleNode>>>`

- `Rc<...>` is an **owning strong reference**. Cloning it increases the strong count and keeps the value alive.
- `Weak<...>` is a **non-owning reference**. Cloning it does not increase the strong count, so it does not keep the value alive.
- `Weak` must be converted using `upgrade()` to get `Option<Rc<...>>` before use.

`Weak` is typically used for `prev` in doubly linked lists to avoid reference cycles and memory leaks.

d- What this line achieves

```
if let Some(ref mut x) = *a.borrow_mut() {(*x).prev = Rc::clone(&b);}
```

This line: - mutably borrows the contents inside `a` (`RefCell`) - checks that `a` contains a node (`Some(...)`) - gets a mutable reference to that node as `x` - sets `x.prev` to point to `b`

So it connects node `a` back to node `b`, completing the backward link (and creating a cycle with `b.next -> a`).