Day 05

Parsing naughty-list.csv

The Story

The elves stared at their screens. They had just written the `Kid` struct and were testing it with Santa's data.

But something was wrong with the data, Prancer leaned back, smirking. "We forgot something obvious, didn't we? The data's raw strings—we need to parse it first."

```
Alice,10,2
Bob,5,5
Charlie,1,9
```

The Story

"We need to create another function," Prancer continued. "to parse the CSV rows into `Kid` structs."

Blitzen slammed his mug down. "And since Santa put me in charge of this project, I'm naming the function. It's going to be called `parse_row`".

The Story

An elf from the back muttered just loud enough to hear, "Ugh, he thinks he's better than us because Santa made him lead."

Blitzen shot them a look. "I heard that. If you've got a better name, I'm listening."

Silence.

"Exactly. `parse_row` it is."

The Frustation

Blitzen paced. "We need a function that takes a CSV row, splits it, and converts it into a `Kid`. Name is easy—it stays a `String`. The good and bad deeds, though, need to be parsed to `u32`."

The Frustation

"But what if the row has garbage data?" asked an elf, holding up a note with `Charlie,,9` scribbled on it.

Prancer rolled his eyes. "Obviously, we handle errors. No `.unwrap()` shortcuts."

The Task

Blitzen wants you to create an associated function for the `Kid` struct and name it `parse_row`. It should take a CSV row as a `&str` and return a `Result<Kid, &'static str>`.

The Task

The function should:

- Split the CSV row into parts.
- Extract the name as a `String`.
- Parse the second and third fields as `u32` for good and bad deeds.
- Finally create a `Kid` struct using the `new()` associated function we created earlier.

Hints

If you're stuck, here are some hints to help guide you:

- **Split the Row:** Use `split(',')` to divide the CSV row into parts. `let fields = row.split(',');`
- **Get Next Field:** Get the next field with `next()`, it's going to return an `Option<&str>`.

Hints

- **Must be mutable:** The `next()` method requires a mutable reference to the iterator. So make it mutable, `let mut fields = row.split(',');` .
- **Transform Option to Result:** Use `ok_or(&str)` to convert the `Option` to a `Result` . e.g., `fields.next().ok_or("Missing field")` .

Hints

- **Propagate Errors (optional):** Use `?` to propagate errors. e.g., `fields.next().ok_or("Missing field")?` .
- **Create a String:** After you get access to the `&str` use the `to_string()` method to make it a `String` and have Ownership.

Hints

- **Parse Numbers:** Parse the second and third fields as `u32` for good and bad deeds. Use `.parse()`, you can either turbofish `parse::<u32>()` or assign a type to the variable `let good_deeds: u32 = fields.next().ok_or("Missing field")?.parse();`.

Hints

- **Map the Error:** The error from the `parse()` method can't be propagated directly, you need to map it to the return types error type `&'static str` using `map_err()`. e.g., `fields.next().ok_or("Missing field")?.parse().map_err(|_| "Invalid good deeds")?`.

Hints

- **Create the `Kid`:** Pass the extracted values to `Kid::new` to build the `Kid` struct.
- **Return a `Result`:** Use `Ok` for success and meaningful error messages (like `"Invalid good deeds"`) for failures.