Assignment #1: Rust Introduction

Overview

The goal of this assignment is to gain some experience with various aspects of the Rust programming language. The assignment will not explore every feature of Rust, but should expose you to an interesting subset of the features that you may use in later assignments.

Given Code

You should start by downloading the given file from the course website. You will find a small set of tests (more may be used in grading) for each part of the assignment as well as some starter code for some parts.

To begin, create a new Rust package named `hw1` by running the following command. Then copy the files from each subdirectory in the provided sources into the new package directory.

```
cargo new hw1 --lib
```

You will need to edit `Cargo.toml` to add the following (at the bottom of the file).

```
[dev-dependencies]
time-test = "0.2.1"
```

Part 1: Setup Verification

IMPLEMENT THIS PART IN `src/part1.rs`.

First, let's verify that everything is set up properly to build and test.

Edit the `src/part1.rs` file and define a function `double` that takes an `i32` value and returns the result of doubling that value.

Now you should test this function.

The following *will not yet work* because there are test files that require functionality that you are unlikely to have implemented at this point. That said, the following command will attempt to run all test cases.

```
cargo test
```

At this time, however, you should attempt to test only this part (and similarly as you progress through the assignment) by using the following command to run only those tests in `tests/part1.rs`.

```
cargo test --test part1
```

If all is well, then add more tests (though this is a very simple function) to see how to interface with the testing framework.

Part 2: Vectors and Tuples

IMPLEMENT THESE PARTS IN `src/part2a.rs`, `src/part2b.rs`, AND `src/part2c.rs`.

This part introduces some of the commonly used data structures provided by the Rust language and library.

a) Write the `unzip_ints` function taking a reference to a vector slice of integer pairs (i.e., `&[(i32,i32)]`) and returns a pair of vectors (i.e., `(Vec<i32>,Vec<i32>)`): the first vector contains the first element of each tuple in the input vector and the second vector contains the second elements (preserving the order of the original vector).

   Note: This is a pretty standard unzip, but what is of interest here is expressing this in Rust and understanding the types involved. In particular, without changing the test cases, explore what happens if the argument type for this function is changed to `&Vec<(i32,i32)>`.

   You can test this part by running the following command (and similarly for the next parts).

   ```
   cargo test --test part2a
   ```

b) Copy your `unzip_ints` function as a new function named `unzip_mixed_vec` and change the `i32` in the second position of the tuples (both in the input and the return type) to `String`. Build the program (you can use `cargo check` to run only the checkers) and observe what happens.

The `String` values are owned by the input collection (by the tuple owned by the slice). Our code can borrow these, but it cannot move them (allowing such would effectively invalidate the input slice, but that was borrowed and should not be invalidated). So let's make a change to the type.

Change the argument type of `unzip_mixed_vec` to `Vec<(i32,String)>` such that the function takes ownership of the vector and make any (minor) modifications necessary to pass the tests.

c) The prior function works, but requires ownership of the input vector. Moreover, passing a slice actually requires building a fresh vector from the slice first.

If we do not want the function to take ownership of the vector (and want to support slices), then there are a few options. Only one is considered here.

Copy the `unzip_mixed_vec` function and name the new version `unzip_mixed_slice`. Update the parameter to take a slice as an argument and update the return type such that the second element of the tuple is a vector holding `String` references (i.e., the return type should be `(Vec<i32>, Vec<&String>)`). Make any (minor) modifications necessary to pass the tests.

Part 3: Structures and Maps

IMPLEMENT THIS PART IN `src/part3.rs`.

The provided `data` module includes the definition of a `Ticket` structure. Write a function named `tally_tickets` that takes a slice of `Ticket` values and returns a `HashMap<&str,u32>` mapping each event identifier to the number (`u32`) of tickets taken for that event (based on the tickets in the input slice).

Note: you can `use super::data::Ticket` to bring the structure into the local namespace.

Part 4: Options, Results, and Matches

IMPLEMENT THIS PART IN `src/part4.rs`.

This part explores multiple features at once, so the function that you will write is (intentionally) a bit peculiar.

Write a generic function (i.e., parameterized on `T`, where `T` must implement `Eq`) named `find_unique`. This function must take a value of type `&[Option<T>]` (i.e., a borrowed slice of `T` options) and a borrowed `T` value.

The function must search the slice for the provided value (only checking those elements for which there is `Some` value). If the value is found in only one position, then the function returns an `Ok` result with the index at which the value was found. Otherwise, the function returns an `Err` result with a vector of all indices at which the value was found.

There is a lot going on, so here is the intended function header (if this incantation does not make sense, let's discuss it).

```
pub fn find_unique<T>(
   haystack: &[Option<T>],
   needle: &T
) -> Result<usize, Vec<usize>>
where
   T: Eq,
```

You should consider the use of `Iterator` and `Enumerate` via `.iter()` and `.enumerate()`. You are also encouraged to use `match` for handling the `Option` simply to learn the construct (though an `if let` is reasonable here).

Part 5: Simple Lists – Heap Allocation

IMPLEMENT THESE PARTS IN `src/part5a.rs` AND `src/part5b.rs`.

In `src/data.rs` there is a declaration of a `List<T>` trait. You will find implementations of this trait in `src/part5a.rs` and `src/part5b.rs`. Read these implementations and note the similarities. Now run the tests (`cargo test --test part5 -- --test-threads=1`) and note the relative run times for each test. Read the implementations again and consider the cause of these differences.

Now move on to the implementation aspects of this part.

a) Uncomment the declaration of `append` in the `List<T>` trait in `src/data.rs`.

b) In `src/part5a.rs`, implement the `append` trait method for the `BoxList` type such that the result of append is a `BoxList` with all of the elements of the "self" list followed by all of the elements of the parameter list. Avoid the use of the `tl` and `cons` methods in your implementation (ideally the reason for this is apparent after the testing example above, but let's discuss this).

c) In `src/part5b.rs`, implement the `append` trait method for the `RcList` type such that the result of append is a `RcList` with all of the elements of the "self" list followed by all of the elements of the parameter list. Should you avoid the use of `tl` or `cons` for this implementation?

For a more detailed and interesting exploration of lists (primarily as a tool for learning Rust), consider reading through Learn Rust With Entirely Too Many Linked Lists.

Part 6: Closures

Implement this part in `src/part6.rs`.

Reimplement the function named `find_unique` from Part 4, but this time use an iterator and higher-order functions such as `filter` and `map`.

Part 7: Arithmetic Language

Implement this part in `src/part7.rs`.

Examine `src/arith/expr.rs` and `src/arith/build.rs` to see the data definition of a small arithmetic/boolean language (based on the material covered in the textbook and presented in lecture) and a set of functions to aid in the construction of arithmetic/boolean expressions.

Write a function named `gather_nums` that takes an expression (technically a `&Rc<Expression>` and returns of vector of all `i32` values within the expression.

Part 8: Completing a Small-Step Evaluator for Arithmetic Language

Implement this part by modifying `src/eval.rs`.

As in the previous part, examine `src/arith/expr.rs` and `src/arith/build.rs` to see the data definition of a small arithmetic/boolean language (based on the material covered in the textbook and presented in lecture) and a set of functions to aid in the construction of arithmetic/boolean expressions.

For this part, you will complete the implementation of the single-step evaluator in `src/eval.rs`. More specifically, you will need to complete the implementation of the evaluation of `sub` and `if` expressions. The small-step semantics evaluation rules are given on the last page where the set of values consists of the boolean values (`true` and `false`) and the numeric values (integers as represented in an `i32`).

Grading

Grading will be divided as follows.

Part	Percentage
1	5
2	15
3	10
4	10
5	15
6	10
7	10
8	25

The terms for the expression language are to be inferred from the rules below coupled with the discussion in lecture (and in the textbook). Your implementation, of course, will work on the internal AST representation of such expressions, so you should be able to map the terms used in the evaluation rules to the variants declared in the Rust code.

nv is for numeric values.

$$\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \qquad \text{(E-If)}$$

$$\text{iszero } 0 \longrightarrow \text{true} \qquad \text{(E-IsZeroZero)}$$

$$\frac{nv_1 \neq 0}{\text{iszero } nv_1 \longrightarrow \text{false}} \qquad \text{(E-IsZeroNonZero)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{iszero } t_1 \longrightarrow \text{iszero } t_1'} \qquad \text{(E-IsZero)}$$

$$nv_1 + nv_2 \longrightarrow nv_1 +_{int} nv_2 \qquad \text{(E-AddConst)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2} \qquad \text{(E-AddLeft)}$$

$$\frac{t_2 \longrightarrow t_2'}{nv_1 + t_2 \longrightarrow nv_1 + t_2'} \qquad \text{(E-AddRight)}$$

$$nv_1 - nv_2 \longrightarrow nv_1 -_{int} nv_2 \qquad \text{(E-SubConst)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 - t_2 \longrightarrow t_1' - t_2} \qquad \text{(E-SubLeft)}$$

$$\frac{t_2 \longrightarrow t_2'}{nv_1 - t_2 \longrightarrow nv_1 - t_2'} \qquad \text{(E-SubRight)}$$