Little Languages

Lecture 27:

Operator Overloading

VM day today!
Update Rust: $ rustup update
Pull 590-material from upstream.

Let's Implement a Rational Number Module

- A rational number is made of two integers: numerator, denominator

$$\frac{\text{numerator}}{\text{denominator}}$$

- Arithmetic operators can be applied to rational numbers:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2},$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2},$$

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2},$$

$$\frac{n_1 / d_1}{n_2 / d_2} = \frac{n_1 d_2}{d_1 n_2},$$

Warm-up: Simplify Rationals

- Open:
  - 27-operator-overloading/src/main.rs
  - 27-operator-overloading/src/rational.rs
- Notice two Rational objects are constructed and printed...
- ...they're not simplified, though!
- Fix the constructor so that all Rational objects are simplified upon construction.
- Check-in on PollEv.com/compunc when complete and then think through how you would write an add method to add two Rationals together.

Operators on Complex Data Types

- Some data types are well suited to use operators like +, -, *, /:
  - Rational numbers
  - Vectors in the mathematical sense
  - Matrices
  - Data tables
- Relational operators like ==, <, >, etc. are commonly useful, as well
- Most programming languages you've used do not allow you to extend the meaning of operators dependent on their usage
  - For example, to test equality of Strings in Java you say s1.equals(s2) ... *yuck*

Operator Overloading

- Some languages allow you to define the meaning of operators on user defined types:
  - C++, C#, Python, Rust, Ruby, and many others

- Suppose you're defining a type **T** and have two objects **a** and **b** of type **T**
  - In your programs you'd like to be able to write: **a + b ...** how is this made possible?

- General strategy for operator overloading:

1. You add specifically named and typed methods to your data type T
2. When the compiler reaches an addition expression LHS is of type T, it
   - Looks to see if T has the specially named method defined on it. If not, error.
   - If so, substitute a + b with a.specialMethod(b)
   - This idea of "magic method calls" is pervasive with *toString* methods even in Java

- Each language that supports operator overloading has its own conventions for implementing.

Operator Overloading in Rust

- We'll take a high-level pass at operator overloading. Full detail in Ch 12.
- Many operators can be overloaded. The book's table 12-1 (right) is a great reference.
- Each operator you want to overload has its own trait. You must implement this trait for the left-hand side's type.

Table 12-1. Summary of traits for operator overloading

Category	Trait	Operator
Unary operators	std::ops::Neg	-x
	std::ops::Not	!x
Arithmetic operators	std::ops::Add	x + y
	std::ops::Sub	x - y
	std::ops::Mul	x * y
	std::ops::Div	x / y
	std::ops::Rem	x % y
Bitwise operators	std::ops::BitAnd	x & y
	std::ops::BitOr	x | y
	std::ops::BitXor	x ^ y
	std::ops::Shl	x << y
	std::ops::Shr	x >> y
Compound assignment arithmetic operators	std::ops::AddAssign	x += y
	std::ops::SubAssign	x -= y
	std::ops::MulAssign	x *= y
	std::ops::DivAssign	x /= y
	std::ops::RemAssign	x %= y
Compound assignment bitwise operators	std::ops::BitAndAssign	x &= y
	std::ops::BitOrAssign	x |= y
	std::ops::BitXorAssign	x ^= y
	std::ops::ShlAssign	x <<= y
	std::ops::ShrAssign	x >>= y
Comparison	std::cmp::PartialEq	x == y, x != y
	std::cmp::PartialOrd	x < y, x <= y, x > y, x >= y
Indexing	std::ops::Index	x[y], &x[y]
	std::ops::IndexMut	x[y] = z, &mut x[y]

Follow-along: Overload the Multiplication Operator

- The multiplication operator's trait is Mul
- Let's implement it for Rational as shown below
- Notice the mul method's self is the left-hand side rational and the right-hand side rational is the second parameter of the method.
- Output is the associated type specifying the return type of the operator.

```rust
impl Mul for Rational {
    type Output = Rational;
    fn mul(self, rhs: Rational) -> Rational {
        Rational::from(self.n * rhs.n, self.d * rhs.d)
    }
}
```

- Now, in main, let's try multiplying our two Rationals together.

Hands-on: Implement the Addition Operator

- Add another impl block Add for Rational.
  - It should look exactly like Mul's except the function's name is add.
- Implement the arithmetic to return a Rational that's: lhs + rhs
- Try using the addition operator in main to test its correctness.
- Check-in when your overloaded addition is working.

```rust
impl Add for Rational {
    type Output = Rational;
    fn add(self, rhs: Rational) -> Rational {
        Rational::from(
            self.n * rhs.d + rhs.n * self.d,
            self.d * rhs.d,
        )
    }
}
```

Follow-along: Operating on Different Types

- What if we wanted to be able to add an i64 with a Rational?
- The default impl of traits assumes the same type for LHS and RHS.
- You can override the RHS with a generic type on the Trait. For example:

```rust
impl Add<Rational> for i64 {
    type Output = Rational;
    fn add(self, rhs: Rational) -> Rational {
        Rational::from(self, 1) + rhs
    }
}
```

Hands-on: Addition for Rational + i64

- Add another impl block Add for Rational.

- Instead of overloading addition for i64 + Rational it should overload for Rational + i64.

- Come up with an example to test in main.

- Check-in when your code is working!

Preview: Implementing a Macro

- We currently construct Rationals via the Rational::from static method
- For example: **`Rational::from(1, 2)`**
- Wouldn't it be nice if we could express a Rational more naturally?
- Perhaps something like: **`rat!(1 / 2)`**
- With a *function* this is generally impossible because the 1 / 2 expression is evaluated *before* the "rat! function" would be called.
- With a *macro*, because macros are *expanded* in an early stage compilation, we can match against the three tokens (1, /, 2) and *rewrite* a substitution *using* those *tokens*.

Defining a simple macro

- Macros *preprocess* your source code to make substitutions *before* compilation
  - They're a deep subject with *lots* of nuances covered in Chapter 20

- To make *any* sense of macros requires understanding *tokens* and *parse trees*

- In Rust, a macro definition specifies patterns of tokens or AST nodes to match
  - Those tokens / AST nodes are then substituted into a template of Rust code

- e.g. the rules below match lhs/rhs "**t**oken **t**rees" separated by a "/" token

```
#[macro_export]
macro_rules! rat {
    ($lhs:tt / $rhs:tt) => (Rational::from($lhs, $rhs))
}
```