Algebraic Data Types and Patterns

16 October 2026

Instructions

Your task is to implement a small computer algebra system (CAS), i.e. a simple Wolfram Alpha. In particular, it will support computing derivatives and simple simplification of expressions.

1. Download the starter code file.
2. Implement the functionality described in the next section.
3. Check that the program passes the tests (`cargo test`).
4. Fix all compiler warnings.
5. Find common problems in the code using `cargo clippy` and fix them (problems with the method name `to_string` may remain).

Functionality

1. Add an enum `Var` with the possible values `X`, `Y` and `Z`. It will represent variables. Add `derive(Copy, Clone, Debug, PartialEq)`. The last one allows comparing two objects of type `Var` with `==`.
2. Implement a method `to_string(&self) -> String` for `Var`. Use `match`.
3. Add an enum `Const` with the possible values `Numeric(i64)` and `Named(String)`. It will represent numeric and named constants.
4. Implement a method `to_string(&self) -> String` for `Const`. Use `match`.
   Add `derive(Clone, Debug)`.
5. Implement an enum `E` (for Expression, but a short name will be more convenient for us here) that will represent symbolic expressions. Its values are:
   (a) `Add(Box<E>, Box<E>)` (addition),
   (b) `Neg(Box<E>)` (negation),
   (c) `Mul(Box<E>, Box<E>)` (multiplication),
   (d) `Inv(Box<E>)` (inverse),
   (e) `Const(Const)`,
   (f) `Func { name: String, arg: Box<E>}` (named functions),
   (g) `Var(Var)`.

   Add `derive(Clone, Debug)`. What are `Const(Const)` and `Var(Var)`? Can we replace `Box<E>` with just `E`? Why?

6. For each variant of `E`, implement a constructor as an associated function of `E` that takes the expression's argument(s) as `Box<E>`, creates the corresponding object and wraps it in a `Box`. For example, the signature of the addition constructor will look like this: `fn add(Box<Self>, Box<Self>) -> Box<Self>`. You can see calls to these constructors in the tests (`test_builder_*`). They make it easier to build complex expressions without constantly calling `Box::new`.
7. Add a method `to_string(&self) -> String` to `E`. Use `match`.
8. Add a method `arg_count(&self) -> u32` to `E` that returns the number of arguments of the root of the expression. For example, for 5 it returns 0, for $f(3 + X)$ it returns 1, and for $3 + g(5)$ it returns 2. Use a `match` with **at most three arms**.
9. Add a method `diff(self, by: Var) -> Box<Self>` to `E` that computes the derivative of the expression `self` with respect to the variable `by`. Some requirements and hints:
   (a) a recursive implementation,
   (b) the method implemented entirely as a single `match`,
   (c) the derivative of a function `f` with respect to the variable `X` is denoted `f_X`,
   (d) an example `match` arm:
   `Self::Add(e, e1) => Self::add(e.diff(by), e1.diff(by))`.
10. Add a method `unpack_inv_inv(self) -> Option<Box<Self>>` to `E` that returns e for an expression of the form $1/(1/e)$, and `None` for other expressions. **The implementation should use two refutable assignments**.
11. Add a method `uninv(self: Box<Self>) -> Box<Self>` to `E` that returns the expression `self` with double inverses removed, using `while let` to call `unpack_inv_inv` repeatedly. If the expression does not have this form, it is returned unchanged.
12. Add the methods `unpack_neg_neg(self) -> Option<Box<Self>>` and `unneg(self: Box<Self>) -> Box<Self>` to `E`. They work exactly like the methods in items 10 and 11, but `unpack_neg_neg` is implemented with `if let ... && let ...`.
13. Add a method `substitute(self, name: &str, value: Box<Self>) -> Box<Self>` to `E` that substitutes `value` for every occurrence of the constant `name` in `self`. Use `match` for the implementation, with **no more than two arms** handling the cases with `E::Const`.
14. Use the implemented system in the `main` function in any way, getting rid of warnings about unused functions and types.

Requirements

- The code compiles with the stable Rust compiler.
- No warnings from the compiler or `clippy` (problems with the method name `to_string` may remain).
- All tests pass.
- Full implementation of the specified functionality.

Grading (3 pts)

- 1 pt: Work during the laboratory.
- 1 pt: Full functionality (complete implementation meeting the requirements).
- 1 pt: Presentation of the solution and answers to the instructor's questions.