Algebraiczne Typy Danych i Wzorce

22 października 2026

Instrukcja

Twoim zadaniem jest implementacja małego systemu algebry komputerowej (CAS), czyli prostego Wolframa Alpha. W szczególności będzie wspierał obliczanie pochodnych i proste upraszczanie wyrażeń.

1. Pobierz plik z kodem startowym.
2. Zaimplementuj funkcjonalność opisaną w kolejnej sekcji.
3. Sprawdź, czy program przechodzi testy (`cargo test`)
4. Napraw wszystkie ostrzeżenia kompilatora.
5. Znajdź powszechne problemy z kodem używając `cargo clippy` i je napraw. (mogą zostać problemy z nazwą `to_string` dla metod).

Funkcjonalność

1. Dodaj typ wyliczeniowy `Var`, o możliwych wartościach `X`, `Y` i `Z`. Będzie on reprezentował zmienne. Dodaj `derive(Copy, Clone, Debug, PartialEq)`. Ten ostatni pozwoli na porównywanie dwóch obiektów typu `Var` przez `==`.
2. Zaimplementuj dla `Var` metodę `to_string(&self) -> String`. Użyj `match`.
3. Dodaj typ wyliczeniowy `Const` o możliwych wartościach `Numeric(i64)` i `Named(String)`. Będzie on reprezentował stałe liczbowe i nazwane.
4. Zaimplementuj dla `Const` metodę `to_string(&self) -> String`. Użyj `match`.
   Dodaj `derive(Clone, Debug)`.
5. Zaimplementuj typ wyliczeniowy `E` (od Expression, ale krótka nazwa będzie dla nas tutaj wygodniejsza), który będzie reprezentował symboliczne wyrażenia. Jego wartości to:
   (a) `Add(Box<E>, Box<E>)` (dodawanie),
   (b) `Neg(Box<E>)` (negacja),
   (c) `Mul(Box<E>, Box<E>)` (mnożenie),
   (d) `Inv(Box<E>)` (odwrotność),
   (e) `Const(Const)`,
   (f) `Func { name: String, arg: Box<E>}` (nazwane funkcje),
   (g) `Var(Var)`.

   Dodaj `derive(Clone, Debug)`. Czym jest `Const(Const)` i `Var(Var)`? Czy możemy zastąpić `Box<E>` po prostu `E`? Dlaczego?

6. Dla każdego wariantu `E` zaimplementuj konstruktor będący funkcją skojarzoną `E`, który przyjmuje argument(y) wyrażenia jako `Box<E>`, tworzy odpowiedni obiekt i pakuje go w `Box`. Dla przykładu, sygnatura konstruktora dodawania będzie wyglądać tak:
`fn add(Box<Self>, Box<Self>) -> Box<Self>`. W testach (`test_builder_*`) możesz zobaczyć wywołania tych konstruktorów. Ułatwią one budowanie złożonych wyrażeń bez konieczności ciągłego wołania `Box::new`.
7. Dodaj do `E` metodę `to_string(&self) -> String`. Użyj `match`.
8. Dodaj do `E` metodę `arg_count(&self) -> u32`, która zwraca liczbę argumentów korzenia wyrażenia. Na przykład, dla 5 zwraca 0, $f(3 + X)$ zwraca 1, a dla $3 + g(5)$ zwraca 2. Użyj `match` z **maksymalnie trzema gałęziami**.
9. Dodaj do `E` metodę `diff(self, by: Var) -> Box<Self>`, która oblicza pochodną wyrażenia `self` po zmiennej `by`. Kilka wymagań i wskazówek:
   (a) implementacja rekurencyjna,
   (b) metoda zaimplementowana w całości jako jeden `match`,
   (c) pochodna funkcji `f` po zmiennej `X` oznaczana przez `f_X`,
   (d) przykładowa gałąź `match`:
   `Self::Add(e, e1) => Self::add(e.diff(by), e1.diff(by))`.
10. Dodaj do `E` metodę `unpack_inv_inv(self) -> Option<Box<Self>>`, która dla wyrażenia postaci $1/(1/e)$ zwraca e, a dla innych wyrażeń zwraca `None`. **Implementacja powinna użyć dwóch odrzucalnych przypisań (refutable assignment)**.
11. Dodaj do `E` metodę `uninv(self: Box<Self>) -> Box<Self>`, która zwraca wyrażenie `self` z usuniętymi podwójnymi odwrotnościami przy pomocy `while let` wielokrotnie wywołującego `unpack_inv_inv`. Jeśli wyrażenie nie ma takiej postaci, to jest zwracane bez zmian.
12. Dodaj do `E` metody `unpack_neg_neg(self) -> Option<Box<Self>>` i
`unneg(self: Box<Self>) -> Box<Self>`. Działanie identyczne jak metody w punktach 10 i 11, ale `unpack_neg_neg` jest zaimplementowane przez `if let ... && let ...`,
13. Dodaj do `E` metodę `substitute(self, name: &str, value: Box<Self>) -> Box<Self>`, która w `self` podstawia `value` pod każde wystąpienie stałej `name`. Do implementacji użyj `match`, przy czym powinny być **nie więcej niż dwie gałęzie** odpowiedzialne za obsługę przypadków z `E::Const`.
14. Użyj zaimplementowanego systemu w funkcji `main` w dowolny sposób, pozbywając się ostrzeżeń o nieużytych funkcjach i typach.

Wymagania

- Kod kompiluje się przy użyciu stabilnego kompilatora Rust.
- Brak ostrzeżeń z kompilatora i `clippy` (mogą zostać problemy z nazwą `to_string` dla metod).
- Wszystkie testy przechodzą.
- Pełna implementacja zadanej funkcjonalności.

Ocena (3 pkt)

- 1 pkt: Praca w trakcie laboratorium.
- 1 pkt: Pełna funkcjonalność (pełna implementacja zgodna z wymaganiami).
- 1 pkt: Prezentacja rozwiązania i odpowiedź na pytania prowadzącego.