Computational graph task

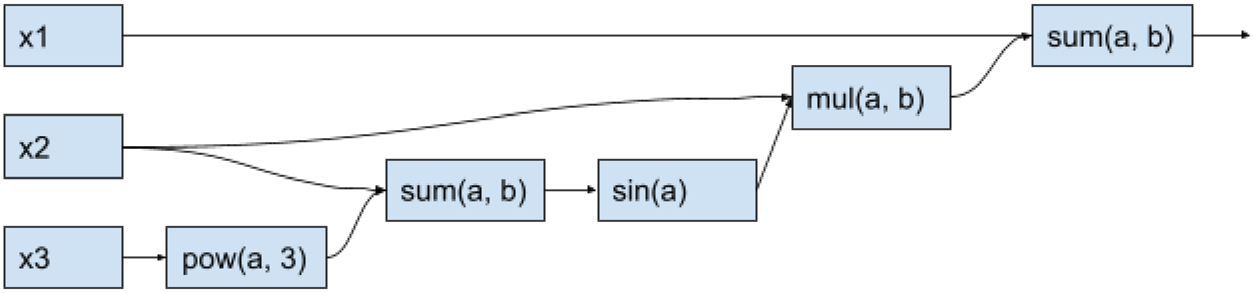
Language: **Rust**

Description:
Computational graph is a direct acyclic graph, where each node takes one or multiple inputs and produces one or more outputs. All inputs and outputs are floating point values (f32).

For example:
*y = x1 + x2 * sin(x2 + pow(x3, 3))*

where *x1, x2, x3* are computation graph inputs and *y* is a graph output. 3 is the exponent for power function.

The corresponding diagram for the above example is the following:



Task:
The task is to create a computational graph structure which can:

1. Take an arbitrary number of inputs
2. Can hold arbitrary graph of operations
3. Can compute a graph with compute() method
4. Each node should have its own cache (the result of **`compute()`** method) and should invalidate ***caches*** of dependent nodes. For example, if *x1* changes in the above example only the last *sum(a,b)* has to be recomputed.

The Rust code for the above example is this:

```
// round to decimal digits
fn round(x: f32, precision: u32) -> f32 {
   let m = 10i32.pow(precision) as f32;
   (x * m).round() / m
}

fn main() {
   // x1, x2, x3 are input nodes of the computational graph:
   let x1 = create_input("x1");
   let x2 = create_input("x2");
   let x3 = create_input("x3");

   // graph variable is the output node of the graph:
   let graph = add(
       x1.clone(),
       mul(
           x2.clone(),
           sin(
               add(
                   x2.clone(),
                   pow_f32(x3.clone(), 3f32)
               )
           )
       )
   );
   x1.set(1f32);
   x2.set(2f32);
   x3.set(3f32);

   let mut result = graph.compute();
   result = round(result, 5);
   println!("Graph output = {}", result);
   assert_eq!(round(result, 5), -0.32727);

   x1.set(2f32);
   x2.set(3f32);
   x3.set(4f32);
   result = graph.compute();
   result = round(result, 5);
   println!("Graph output = {}", result);
   assert_eq!(round(result, 5), -0.56656);
}
```

In the above example functions **`create_input`**, **`add`**, **`mul`**, **`sin`**, **`add`** and **`pow_f32`** creates nodes of a graph.

Notice: because of the *cache* feature, each computational node should maintain a list of nodes that depend on the given one.

We assess your ability to write clean structured code as well as unit tests.