Own language interpreter

I have chosen to write my own language's an interpreter in Rust. The scope of the project was to create a language that is very simple, and basic, then try and run it in some kind of interpreter, written from the ground up.

Before starting I have watched an 8 hours video on writing a Lox interpreter in Rust - Video. This was very useful, it had given me some ideas on how to do the implementation. The Pratt Parser presented proved to be simple to use, and the idea to make the Lexer implement the Iterator trait made things easier in future uses.

The first goal was to create the Lexer. It uses a state machine to determine the current type of the Token. It is easy to extend the functionality, and with error handling it can be further extended with very special cases. For example regular expressions. The implementation works flawlessly.

The next goal was to create the Parser. As said I use the Pratt Parser algorithm. It has many benefits. First of all instead of running multiple passes such as in a Shunting Yard algorithm, we can easily do it in one pass. This reduces runtime, making the language somewhat faster to run. Secondly it is very easy to extend. Simply put, the algorithm determines what expressions should be first calculated, to calculate the next one. When we introduce a new operator we can easily interpret what is the order that the expression should be evaluated. A great example is when I implemented the Boolean operators, and relational operators. It took me only a few seconds to introduce the new operator and set the binding powers, because an "or" operator should be later calculated than "and" operator. The third benefit is that we can instantly jump to an abstract syntax tree and skip the concrete syntax tree.

In more detail, when we run the algorithm it checks the left hand side of the next 2 tokens. If it is an operator, we know that it is an operation where we only have a right hand side value, a prefix operation. If we see a number than we can check the second Token that should be an operator kind. We check if the operator is a postfix operator, or an operator that only has a left-hand side value. If no, then we if it is an infix operator and check its binding power, then for its right hand side value, we call the function it self recursively. If the binding power that the next operator is lower than the passed one, we return the left hand side in the function, the value. I am no teacher, but this source explains it much better than me: Pratt Parser.

The next step is evaluating, and this is the last step that I could complete. I take the values gathered from the Pratt Parser, and do the calculations based on the Operators. It is somewhat easy, but there were some complications, here as well.

In the future I would like to finish this project. The evaluator could use the Iterator trait as well. For the Atomic operations itself, I could create my own traits, or generic functions, to simplify code.

The next step would be to store the variables themselves, and this is the hardest part. There are solutions to it, but it was so much new information that sadly I run out of time. For a very basic language I could use a simple stack-based solution. However, a garbage collector-based solution is something that would make the most sense.

To evaluate the work I did. I learned so many new ideas, and most importantly I fixed some things that I misinterpreted in the past. For example, traits, and lifetimes are totally new to me, and the stack vs heap argument was fully misunderstood by me – the memory allocation is slow, not the reading, or modifying data itself.