Rust Report: Caitlin Goodger 300412363

This program is divided into two parts. First the expression is parsed and then that parsed expression is compared to the target string. The expression parser is recursive decent parser. This means that it is a top down parser. This means that there is no need for backtracking through the expression and the parser runs through the expression string once before returning a parsed expression. Since the | has the widest reach, that is at the top of the parser. The next symbol down is if there is no operator between characters e.g. between a and c in ac. Then the * and the last symbol is a literal character or a dot, which I have grouped together for simplicity. The parser starts at the top and travels down until it matches the current character of the expression. If that character need two arguments, e.g. for the |, then it will continue on to get the second argument before returning an enum of that character type. This process will continue until the entire expression has been parsed.

Once the expression has been parsed, it is then compared to the target by stepping down the expression. There is a match statement to take a different action depending on the enum type encountered. This method will call itself recursively until the program has gone down the entire parsed expression. Once finished, the output is printed. Characters from the target are slowly removed as they match the parsed expression. That means that if there are still characters in the target, then the expression hasn't matched the entire target, e.g. and expression of abc doesn't match a target of abcd. If the target length is 0, then the output is whatever is returned from the matching method. This is the output that the user will see.

One feature of Rust that I really liked was the Result. It allows for the program to easily return errors in the event that something goes wrong. This is really useful when parsing the expression because it means that the program can return an Error if something goes wrong without crashing the entire program or having lots of try catch loops and thrown exceptions. After the expression has been parsed, successfully or not, there is a match statement to check if an error has been returned. If there is an error, SYNTAX ERROR will be printed and if there isn't, then the expression has been parsed correctly. Result makes it an easy way to deal with errors by collecting them all at the end.

I also really like being able to return multiple values. This was particularly useful when matching the target. It allows me to return the target and a YES or NO depending on whether it matched. This meant that I could pass target to a method call, edit the target depending on if it matched the expression before returning the new target so that it could be used for another method call. This approach would not be possible if I was limited only one return value. I would have to decide whether to return the target or the YES or NO. What I would have to do would return the target and then at the end, if there was any target left then print YES or NO. This would make the solution more complicated because it wouldn't know if it matched until the end. There would also be a chance for mistakes if the expression was asking for abcd and the target was abc. When it reached the d in the

expression, the target would already have a length of 0 so it would think that it matches when it doesn't. This issue would be something that would have to be resolved but I didn't have to worry about due to being able to return multiple values.

The memory management in Rust led to some headaches. The biggest issue was around ownership of variables. An example of this is when matching the OR expression to the target. The OR has a left and right and both need to be called to see which one matches the target. Therefore, both need to be passed the target string. In order to be able to pass the target to the right method call, it needs to be cloned in the left side so that the right side still has a copy available. Throughout developing this program, I ran into a number of memory management issues where the variable I was attempting to pass was no longer available in memory since it had been passed in a different method call. Thankfully, all of the issues that I ran into could be solved by cloning the variable. Thankfully, this did not lead to further memory issues and the copy made by the clone was good enough for the program to continue.

The other minor annoyance that I had when creating this program, was the reading in the files and then getting the data out. In the previous assignments, I would read each of the files into an array and then have one for loop to iterate through them. By looping through both of them at the same time, I could ensure that I was always looking at the same line in both files. If I was looking at the expression on line 5 from the expression file, I would definitely be looking at the target string on line 5 from the target file. However, I couldn't do this the same way in Rust. Actually reading in the file was easy and they got read into Lines. The issue was getting the information out of that. The only way I could find to get the data out was to use a for each loop. The issue is that that means that I could only look at one file at once. Therefore, what I had to do was read and parse all the expressions and then add them to a Vector to store them. Once I had parsed all the expressions, I then moved onto reading in the targets. As I looped through the targets, I would remove the first parsed expression from the Vector and compare that to the target. This approach works but there is a chance that the expression and target get out of line and at some point be comparing the expression to the wrong target. Because of how you read in files in Rust, I had to change how I read the files from previous assignments.

The approach that I have taken for this assignment is similar to how I did the F# assignment. For both assignments, I parse the expression before comparing it to the target. In order to store the parsed expression for both assignments I have used enums for the different types of characters. I have slightly changed what enums I have. In the F# assignment, I have a type called End. This was used at the end of a branch of the tree. This was because each enum had a next node so that it knew what the next enum was going to be. So, if it was at the end, the next node would be of type End. This also lead to issues around the * if the previous character was) because it wouldn't know what came inside the loop. There were also issues around the End character when matching it to the target. It could not tell the difference

between the end of target and the end of the side of an or. This led to the program thinking that it didn't match because it thought that it was finished matching when it wasn't. This time I had a Pattern enum that is for a sequencing of characters. This stores a Vector of them so that when it matches it just loops through the list. This makes it much easier as there is no need to store a Next Node like in F# since the Pattern can handle all of that. That makes matching it much easier.

I have also changed how I parse the expression. In the F# assignment, the program had issues matching or statements because of how it parsed. This was due to the fact that it had no order for what symbol it was looking for. It was just one method getting called recursively for the expression string. This meant that It wouldn't consider an | until it found one. By this point, it had already moved passed character on the left side so they would get dropped from the OR, leading to incorrect results. By using a recursive decent parser, it is able to get all the characters in the left side which leads to better results. The comparing the target to the parsed expression is similar across both assignments. The bulk of the work is done within one match statement that matches the expression to the target.

F# and Rust are two very different languages. The theory behind the program is similar but the code is very different. I found Rust much easier to work with than F# because I found that it was easier to manipulate String which was a large part of how I match the target and read the expression. In both languages I had to convert it into an Array but in F#, I was limited it getting the first element or the tail. This could be extremely frustrating at times whereas with Rust that was much easier. The memory management was a new issue that I didn't have to deal with in F#. However, as I mentioned above, it didn't cause me many issues in Rust so it didn't make F# easier in that regard.

The approach that I took in Rust is very different to the approach that I took in Ruby. In Ruby, I matched the target to the expression at the same time I was parsing it. This meant that I had two very different experiences with the languages. Personally, I found Ruby an easier language to program in just because it was more similar to other languages that I had used. However, Rust has a lot of very good documentation so it was easy to pick up. Because my approach to the two assignments was so different, it is very difficult to compare them because I was dealing with very different issues.