There are two main parts to the program. The first part is parsing the expression and then the second part is comparing that to the target. When parsing the expression, the program steps through the expression and forms an Expression Tree using a discriminated union. A discriminated union has a type for each of the symbols that occur in the expression, a literal character, a dot, an asterisk, a pipe or brackets as well as a type for an End character.

As the program parses the expression, it matches the character with a type and branches depending on the type. This means that once the whole expression is parsed, a tree is formed from the expression. Once the expression is parsed, the program then uses the tree to compare it to target. To do that, the program steps down the tree, comparing it to the target at each step. When it gets to a type such as the pipe that has two branches, the program goes down each branch to see which one matches the target.

The reason that I have built it this way was so that I could parse the expression first before comparing to the target. On the previous assignment, I parsed the expression at the same time as target that this lead to issues as the program would fail to match a target that it shouldn't have. So, I changed by approach for this assignment to avoid that issue. Since I was now parsing the expression first, I needed some way to store the expression before comparing to the target. By using a discriminated union, it is easy to store and then step down the tree when comparing it to the target.

Before using the discriminated union, I had attempted to parse the expression differently. Rather than storing the parsed expression in a discriminated union tree, I was sorting it in an Array. It was effective to a point but the Array struggled to had pipes and asterisks. I ran into a number of issues adding to Arrays that were empty and kept getting Null Pointer Exceptions. This was not effective at sorting at the parsed expression so I changed by approach to sorting the expression in a discriminated union which was a better approach.

The discriminated union was an extremely helpful feature of F#. It proved a vague tree structure which was helpful for comparing to the target. It made it easy to handle all the different symbol types in one data type. I also used a number of match expressions which was extremely helpful when both parsing the expression and comparing it to the target. When parsing the expression, I could match it to all the symbol types and take different actions depending on the symbol. The same philosophy could be used when matching it to the target, by matching the type of the discriminated union that the tree was currently at. It was much easier than having numerous if statements.

Since I was stepping through a tree, I also used recursive methods. This means that I could have one method that is used for parsing the expression and looping through that method until reaching the end of the expression. By separating the parsing and matching, it meant that methods were separate and they didn't have to be mutually recursive, saving some headache.

One part of F# that I found irritating, was how it calls methods on Arrays. It wasn't that the way that F# calls methods is wrong, but it is very different to other languages such as Ruby and it took me time to get used to. Rather than calling the method on the array itself, the method is called and then the array is passed in as an argument. It was a different way of calling methods and not something that I had used before.

Another part of F# that annoyed me at times was if statements. In other languages such as Ruby, you can have an if statement without an else clause. In F# if statements have to have an else clause and there were cases that I didn't want an else statement. It meat that I have to occasionally change my approach to separate the program into two parts, one part for the if statement and the other for the else clause. F# also requires that both parts of the statement return the same type of value and there were times that I didn't want to. And that meant that I have to change how I was solving a problem. Both of these issues were fairly minor but took some times to get used to.

When I was testing, I was trying to output to a file, in order to make it easier to compare to the expected output. However, I found this extremely difficult and did not manage to get it working. Since it wasn't required for my program, I moved on but it was interesting how difficult it was, especially compared to Ruby where it was very simple. It is a feature of F# that isn't really required.

Since I have used a different approach than the previous assignment, my F# and Ruby code are very different. There are some similarities in how the programs parse the expression in that it looks at one character at a time, and usually the first character of the expression. But what the program does with that character is very different. The Ruby code immediately compares it to the target, whereas the F# one saves it for later. It meant that my experience with the two languages was very different. However, creating a program in Ruby like I did for F# would have been very different as discriminated unions are a very unique type for F#. This approach is also much easier to extend that the previous version. To add another supported symbol, all that is required is to add it to the discriminated union type, before describing the action to take when parsing it and then when matching. It makes extending the program much easier.