Ryan Darras PJ03 project writeup

Preface

This project was developed using Visual Studio 2017 community edition in C#. There are no customizations made that would interfere with grading or someone else opening and running the project. By default, when the program is run it saves the files into this location ~JackLexer\JackLexer\bin\Debug\Output however I have copied this output into the results folder at the top of the zip file. Similarly, it loads the jack files from ~JackLexer\JackLexer\bin\Debug\JackFiles. The projects starting file is program.cs. I have commented most of the code in the solution to where it shouldn't be too difficult to understand the strategy as you read along.

Abstract

This section of this document will explain the basic details of what is to follow, whereas the more detailed descriptions of each element of this project will be described later. I started this project by bringing over my DFA class from PJ01 so that I could use it to run the DFA's I need in order to tokenize the jack files. The only difference between my NonDeterministicFiniteAutomaton class in PJ01 and PJ03 is that in PJ01 we needed to handle "`" as an epsilon transition, where in PJ03 that is a valid character. Considering this project only needed DFA's and not NFA's, I simply forced my method that gets epsilon transitions to return nothing. My JackTokenizer class works very similarly to my manager in PJ01, such that it loads up all the machines and copies them to progress.

Project

DFA's

To create the DFA's for this project, I simply worked them out by hand similarly to PJ02. I will be including my document that contains some of the more complicated DFA's, although do note that I did go through and make some changes to some of them without making the change in the document, as it was just a reference guide for me when I was debugging through afterwards. Considering many of these DFA's had many transitions in which you had to accept any ascii character, I write a little bit of code that would put all of that in the file, and I would go add the single transitions and the accept state afterwards. Thankfully, I made very few mistakes in my original DFA's that it didn't take too much debugging to find some issues. One thing I did have to troubleshoot was that \r, \n, and \t are single characters, not two characters. I simply saved all of these machines like we did in PJ01 and read them in when I ran the program.

JackTokenizer

This is the class where all the magic happens. In it, I store two strings, one that is the remaining string, and the other is the part that is currently being munched by the current machine. I keep a dictionary of my machines, that are clean and never used, so that I can easily copy from them to run my algorithm. I also keep a dictionary of lists of machines because my PJ01 was set up to accept NFA's and I needed to handle spinning off multiple machines. Each of these lists represent a collection of machines for the specific type of machine that it is. Considering that every machine I have created is a DFA, these lists should only ever be of length 1. (Do note, that I do have 1 or 2 NFA's in the mix as well, but they could be altered to be DFA's with little effort.) The last data structure I use in the JackTokenizer class is another dictionary that contains an integer for every type of machine. I use this to handle the "max munch principle" as described in the project description, as it will maintain how long each machine was able to go and we can use it at the end to get the biggest chunk of string to tokenize. The JackTokenizer has a public method called TokenizeString, that will… as you may have guessed… tokenize the string that we read in from the jack files. This method is what contains the algorithm and the biggest chunk of complexity of this assignment, and I will describe it in the next section.

Algorithm

This algorithm was a stack overflow nightmare to debug through because of the two rather open-ended while loops. However, once I fixed a few small issues in my DFA's, everything seemed to come together fairly easily. It was also easy to find which DFA had the issue because I would run the while loop until it got stuck on something and I would manually read the string to see where it got stuck at. The algorithm works as such:

```
while our string isn't empty:
  loop through each list for each type of machine:
    while we still have a machine in this list processing:
      loop through each machine in the list:
        if machine is on accept state:
          store length of string that got us here.
        advance machine using string[0]
        union results with other machines in list
      remove the first character in the string
      count++
  find the longest segment and store it as a token based on type
  remove the longest segment from our string
  reset our machines to default
```

Our algorithm is relatively complex because of the complexity of our NFA setup. If I rewrote it to only accept DFA’s, it would be a bit simpler of an algorithm. To put the algorithm in layman's terms, it would be something like this:

```
while our string isn't empty:
  foreach machine type:
    temp string = our string
    while machine has not trapped:
      advance using temp string[0], record most recent accept state
      remove temp string[0]
  find the longest result of accepted strings from the machines and store it as a token
  remove the length of this result from our string
  reset machines to be used again
```

Basically we run each DFA on our string and see which one is able to successfully read the farthest. We then take that substring out and tokenize it. Then we run each DFA on the remaining string and rinse and repeat until we don’t have any more string to run through.

Conclusion

I wish I was better at describing algorithms via a document like this, but I did attempt to comment the code in such a way that should make it easy to read through. Overall, I was incredibly confused at the start of the assignment, but once Dr. Bahn replied to my email it all made sense after reviewing his example.