! Proj

Part 1

I am creating a language that parses and approves strings in what is called algebraic chess notation. Algebraic chess notation is a way of tracking moves being played allowing games to be recorded in just plain text. The most basic form of the notation involves the letter of the piece for the first character, then the location of where it will move to.
For example Nf3 means knight to space f3 and e4 means pawn to space e4
Note that these are very basic moves and my language will be able to parse more complex terms such as castling.

I chose this language because chess was a game I played a lot with my family while growing up. It is a game that I enjoy and is very important to me. The chess world championships are currently going on right now which is why it has also been on my mind for a bit.

The language will be represented by an NFA
Alphabet: {[1-8], x, +, #, [a-h], O-O, O-O-O, R, N, B, Q, K}

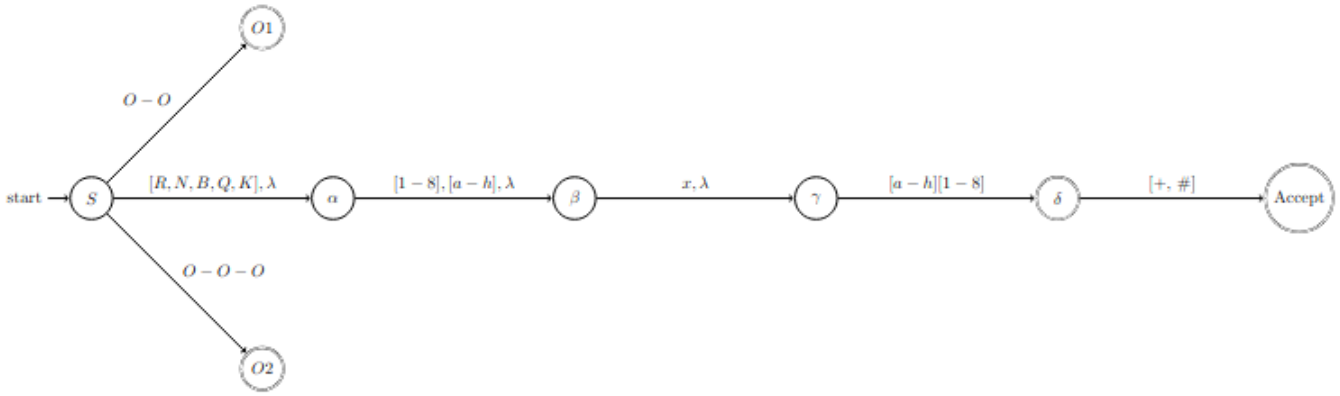
An acceptable string is as follows
([R, N, B, Q, K, λ][a-h, 1-8, λ][x, λ][a-h][1-8][+, #, λ]) | O-O | O-O-O
For a quick breakdown, either you can castle king-side (O-O) or queen-side (O-O-O) or you can make an actual move. An actual move consists of a piece--Rook, kNight, Bishop, Queen, or King (or lambda if its a pawn). Then, if there are multiple pieces that can make the same move, you add either the rank (1-8) or file (a-h) of the piece making the move. After that, if a piece is being captured put an x. After that, add the file and the rank of where the piece will move in that order (mandatory). Finally, if it's a check then put +, if it is checkmate put #, and if it is neither then we are already done.

Part 2 regular grammar

$$
\begin{aligned}
&S \to [R, N, B, Q, K]\alpha \mid \alpha \mid O-O \mid O-O-O \\
&\alpha \to [1-8]\beta \mid [a-h]\beta \mid \beta \\
&\beta \to x\gamma \mid \gamma \\
&\gamma \to [a-h][1-8]\delta \mid [a-h][1-8] \\
&\delta \to + \mid \#
\end{aligned}
$$

Part 3: NFA

Full Automata:



Latex to make:

```
\usetikzlibrary{automata, positioning}

\begin{document}

\begin{tikzpicture}[shorten >=1pt, node distance=4cm, on grid, auto]

\node[state, initial] (S) {$S$};
\node[state] (alpha) [right=of S] {$\alpha$};
\node[state, accepting] (O1) [above right=of S] {$O1$};
\node[state, accepting] (O2) [below right =of S] {$O2$};
\node[state] (beta) [right=of alpha] {$\beta$};
\node[state] (gamma) [right=of beta] {$\gamma$};
\node[state, accepting] (delta) [right=of gamma] {$\delta$};
\node[state, accepting] (end) [right=of delta] {Accept};

\path[->]
(S)    edge node {$[R, N, B, Q, K], \lambda$} (alpha)
    edge node {$O-O$} (O1)
    edge node {$O-O-O$} (O2)
(alpha) edge node {$[1-8], [a-h], \lambda$} (beta)
(beta) edge node {$x, \lambda$} (gamma)
(gamma) edge node {$[a-h][1-8]$} (delta)
(delta) edge node {[$+$, $\#$]} (end);

\end{tikzpicture}

\end{document}
```

Part 4: Data Structure

Alphabet: {[1-8], x, +, #, [a-h], O-O, O-O-O, R, N, B, Q, K}
States: {S, O1, O2. α, β, γ, δ, Accept}
Accepting subset: {O1, O2, δ, Accept}
Start state: {S}

The data structure is represented in my code. Since the automata is essentially linear with options at each, each transition is represented by an if statement for the options. It "parses" by removing the letter that it just accepted. "Jail states" or failed strings that are omitted for simplicity in the NFA but are represented with return False usually in the `else` case of the if statement. Likewise, accepting states are represented with return True

Part 5 Code

This is an explanation of the code (kind of ties into the previous part). Also quick note, while this automata is admittedly simple, it was a lot harder than it looks to implement while keeping the "automata feel". My initial solution worked but it didn't really have states, if that makes sense.

Start state

The start state is represented by the parse function taking in a string and then cleaning it (by calling the strip function)

```
def parse(self, inp):
        inp = inp.strip()
```

O-O, O-O-O

These two symbols represent castling king-side and queen-side respectively. In the NFA, they are represented by two states that branch out of the start state. In the code, they are checked for immediately and if they pass, we return true. Otherwise, we continue parsing down the "main line" of the NFA

```
if inp == 'O-O':
        return True, "Kingside castle"
elif inp == 'O-O-O':
        return True, "Queenside castle"
```

α

The transition from S $\rightarrow \alpha$ involves checking to see if the char is either a piece or lambda if its a pawn. The code does exactly that. It first checks if it is a piece (if it is it goes to the next letter). If it is not then it makes sure the next char is a file because that's the characteristic of a pawn move. If it isn't then we have an invalid move and we return False

```
if inp[0] in self.pieces:
        inp = inp[1::]
elif inp[0] in self.files:
        pass
else:
        return False, "Not a valid piece"
```

β, γ

The transition from $\alpha \rightarrow \beta$ is for piece conflicts. This is when two pieces can make the same move. If this is the case then we need to put the rank or the file but if it isn't the case we can omit it.
The transition from $\beta \rightarrow \gamma$ is to check for an x which denotes a piece was captured. These two transitions were combined into one for my code

```
if inp[0] in self.files or inp[0] in self.ranks or inp[0] in self.takes:
        inp = inp[1::]
else:
        return False, "Not valid rank/file"

# Parse the takes (if there)
if inp[0] in self.takes:
        inp = inp[1::]
```

In my code, we check to see if the first letter of the string is a file or a rank or is an x. If it is, we go to the next letter, if not we return false. After that we do another check for an x the same way. The x is optional so if it is invalid we are still okay

δ

Then, the transition to δ is for adding files (a-h) and ranks (1-8). These are mandatory which is why they are checked for and return false if they fail. Since this is also an accepting state, we check to make sure the length of the input is 0. If it is, then we can accept and are done. Otherwise we continue

```
if inp[0] in self.files:
        inp = inp[1::]
```

```
else:
        return False, "Fails at initial location (file)"


if inp[0] in self.ranks:
        inp = inp[1::]
        if len(inp) == 0:
                return True, "Pass"
else:
        return False, "Fails at initial location (rank)"
```

Accept

The last transition is from $\delta \rightarrow$ Accept. This transition is responsible for checking to see if it is a check (+) or checkmate (#) but is completely optional. It is also the last state so any strings that aren't already finished failed. In my code I check for the end symbol and then check to see if the code passed or not. If it didn't we failed (like jail state)

```
if inp[0] in self.check:
        inp = inp[1::]

if len(inp) == 0:
        return True, "Pass"

return False, "Fails, either not in rank or too many chars"
```

Part 6

As a small little bonus I turned the NFA to a DFA but its messy. (each line is a different option)

S -> O-O | O-O-O |
[R, N, B, Q, K][1-8,a-h]x[a-h][1-8]#
[R, N, B, Q, K][1-8,a-h]x[a-h][1-8]+
[R, N, B, Q, K][1-8,a-h]x[a-h][1-8]

[R, N, B, Q, K][1-8,a-h][a-h][1-8]#
[R, N, B, Q, K][1-8,a-h][a-h][1-8]+
[R, N, B, Q, K][1-8,a-h][a-h][1-8]

[R, N, B, Q, K]x[a-h][1-8]#
[R, N, B, Q, K]x[a-h][1-8]+

[R, N, B, Q, K]x[a-h][1-8]

[R, N, B, Q, K][a-h][1-8]#
[R, N, B, Q, K][a-h][1-8]+
[R, N, B, Q, K][a-h][1-8]

[1-8,a-h]x[a-h][1-8]#
[1-8,a-h]x[a-h][1-8]+
[1-8,a-h]x[a-h][1-8]

[1-8,a-h][a-h][1-8]#
[1-8,a-h][a-h][1-8]+
[1-8,a-h][a-h][1-8]

x[a-h][1-8]#
x[a-h][1-8]+
x[a-h][1-8]

[a-h][1-8]#
[a-h][1-8]+
[a-h][1-8]