| *#Thompson's construction is almost trivial once you have the regex in AST form* *#It is essentially a compiler from the regex AST to NFAs*  *#We already defined the regex AST when we were writing the parser* *#So we need to define the NFA structure before we do anything else*  *#Let's go*  *#So NFAs are simply graphs, any and every way of representing graphs in computer memory can represent them*  *#Let's write the most direct and naive way of representing them*  *#1- State objects to represent nodes* *######### You can add a name string for debugging purposes, but it's not necessary* *######### All we need from a state is to be a placeholder with identity,*  *######### something that is not equal to other instances of its class* class State {}  #2- Edge objects to represent edges (mesh me7taga 7aga ya3ny) class Edge {  State from  State to  set<char|pair<char>> characters }  #3- An NFA is a graph, a bunch of states and edges  class NFA {  #The start and accept states are included in the states array  #We just give them their own reference so they can be fetched in constant time  State start   State accept   #All states of the NFA including the start and the accept  State[] states  Edge[] transitions }  #Now let's write Thompson's algorithm  subroutine Regex-2-NFA of   Input: regex-ast  Output: equivalent-nfa    what is regex-ast.type ? {  *#When we have the regex A|B*   is it an OR node ? {  *#Then assuming we have the NFAs representing A and B*  left-nfa = Regex-2-NFA(regex-ast.left-ast)  right-nfa = Regex-2-NFA(regex-ast.right-ast)   *#Make 2 new states, those are the new start and accept states of the overall NFA*  start = new State   accept = new State    *#The new start goes to each of the 2 start states of A's nfa and B's nfa via an epsilon transitions*  start-transition-1 = new Edge(start,left-nfa.start,{})  start-transition-2 = new Edge(start,right-nfa.start,{})  *#And the new accept has incoming epsilon transitions from the 2 accept states of A's and B's nfa*  final-transition-1 = new Edge(left-nfa.accept,accept,{})  final-transition-2 = new Edge(right-nfa.accept,accept,{})   *#The spread operator '\*' appends all elements of a list into another list*  *#For example, if x = [1,2] and y = [3,4] then z = [\*x,\*y] is equal to [1,2,3,4]*  states = [  \*left-nfa.states,  \*right-nfa.states,   start,  accept  ]   transitions = [  \*left-nfa.transitions,  \*right-nfa.transitions,  start-transition-1,start-transition-2,  final-transition-1,start-transition-2  ]  return new NFA(start,accept,states,transitions)  }  *#When we have the regex AB*  is it a SEQ node ? {  *#Then assuming we have the NFAs representing A and B*  left-nfa = Regex-2-NFA(regex-ast.left-ast)  right-nfa = Regex-2-NFA(regex-ast.right-ast)   states = [  \*left-nfa.states,  \*right-nfa.states  ]   transitions = [  \*left-nfa.transitions,  \*right-nfa.transitions,  *#Simply connect the accept of A's nfa to the start of B's nfa, that's it*  new Edge(left-nfa.accept,right-nfa.start,{})  ]  return new NFA(left-nfa.start,  right-nfa.accept,  states,transitions)   }  *#When we have the regex A\* or A+*  is it a KleeneStar or KleenePlus node ? {  *#Then assuming we have the NFAs representing A*  left-nfa = Regex-2-NFA(regex-ast.left-ast)   *#Make 2 new states, those will be the start and accept of the overall nfa*  start = new State   accept = new State   *#The new start goes to the start of A's nfa (so that it accepts everything A will accept)*  *#It also goes to the new accept directly to represent accepting on empty inputs*   start-transition-1 = new Edge(start,left-nfa.start,{})  start-transition-2 = new Edge(start,accept,{})   *#The new accept has an incoming edge from the accept state of A's nfa*   *#The accept state of A's nfa also loops back to the new start in order to accept more A*  final-transition-1 = new Edge(left-nfa.accept,accept,{})  final-transition-2 = new Edge(left-nfa.accept,start,{})   states = [  \*left-nfa.states,  start,accept  ]   transitions = [  \*left-nfa.transitions,  start-transition-1,final-transition-1,final-transition-2  ]  *#Only add the transition that accept 0 inputs if it's KleeneStar*  if regex-ast.type is not KleenePlus {  transitions.push(start-transition-2)  }  return new NFA(start,accept,states,transitions)  }  *#When we have the regex A?*  is it an Optional node ? {  *#Then assuming we have the NFAs representing A*  left-nfa = Regex-2-NFA(regex-ast.left-ast)   start = new State   accept = new State   *#Do the same as A\* except with no looping transition*  *#Only the transition that accepts without any input*  start-transition-1 = new Edge(start,left-nfa.start,{})  start-transition-2 = new Edge(start,accept,{})   final-transition-1 = new Edge(left-nfa.accept,accept,{})   states = [  \*left-nfa.states,  start,accept  ]   transitions = [  \*left-nfa.transitions,  start-transition-1,start-transition-2,final-transition-2  ]  return new NFA(start,accept,states,transitions)  }  *#When we have the regex 'x' for any character x or a character class [...]*  is it a Literal or a CharacterClass ? {  chars = {regex-ast.c} if regex-ast.type is Literal   else regex-ast.chars  *#The nfa that accepts this has 2 states, one starting and one accepting*  start = new State   accept = new State   *#And a single transition that goes from the starting to the accepting on the relevant characters*  trans = new Edge(start,accept,chars)  return new NFA(start,accept,  [start,accept],  [trans])  }  } |
| --- |