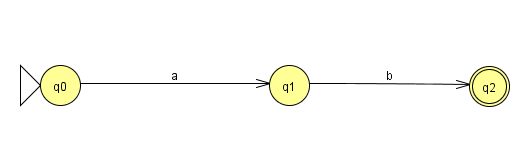
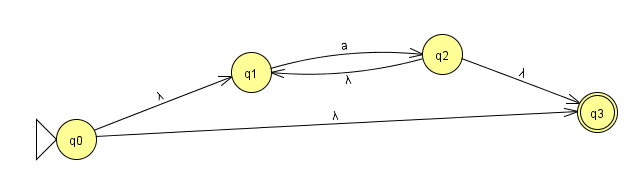
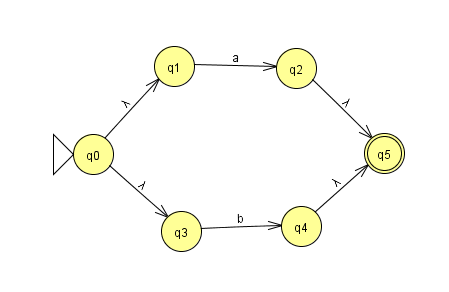


**Converting Regex to NFA**

(ab)



(a|b)

(a)\*

Easiest Round-off Trick!!!:

**How to tell if a regex is regular**

Step 1: Look at the exponents of the different alphabets.

Step 2: Is there any relation between the different exponents?

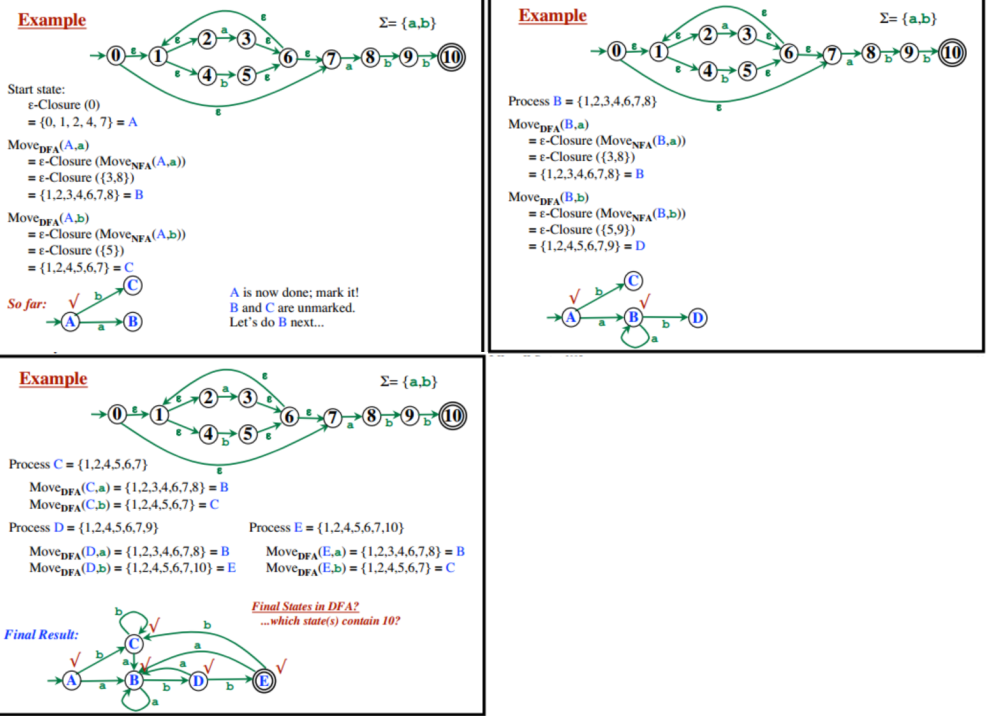
Step 3a: If there is NO relation between the exponents then the language is Regular and Context free. eg: L= *anbm* , m,n >=0

Step 3b: If there are relations between the exponents, for instance, *L*=*ambn*, and the relation can be m>n or m!=n or m< n. You will need one counter here to keep a count, inrementing it by 1 for n times for a and compare with b by decrementing by 1 for n-times and you get a 0 for a match(assuming, m=n). These languages are Not Regular but context free and accepted by a PDA.

Step 4: If you have more than one relation or need more than one counter, for instance, *L*=*ambnck*, m=n=k. Here, you need 2 such counters. First, count all a's and copy this count value to another counter. Then, compare with b by decrementing it n-times to get a 0. Again, decrementing the other counter n-times to compare it with c and accept the language if both the counters are 0. These languages are neither Regular nor Context Free but they are Context Sensitive and thus Recursive, although, all Recursive languages are not Context Sensitive

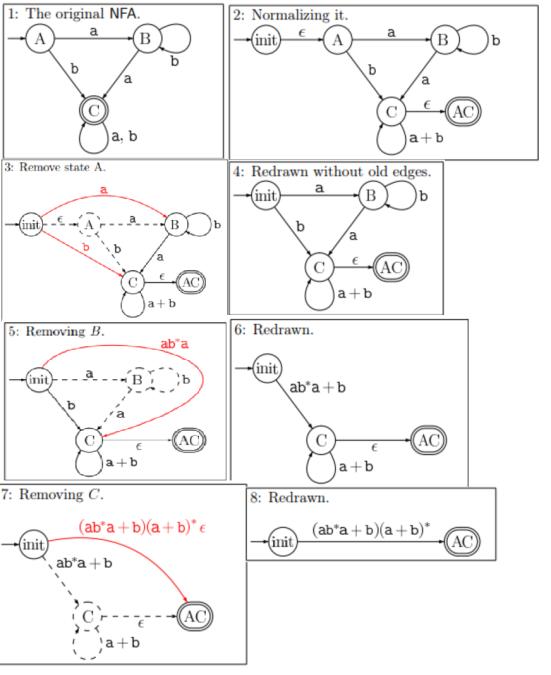
**JFLEX**

Java Regex



Scanner myscan = new Scanner(System.in); long myLong = myscan.nextLong();

StringTokenizer st = new StringTokenizer("This is for Sabahs Bonus!"); while (st.hasMoreTokens()) {System.out.println(st.nextToken());}

mystring = reader.readLine(); token=new StreamTokenizer((new StringReader(mystring)));int numOfTokens = token.nextToken(); while(numOfTokens!= token.TT\_EOF){ System.out.println(token.sval); numOfTokens = token.nextToken();}

String string = "This is-an example"; String[] str = string.split("-"); System.out.println(str[0]);System.out.println(str[1]);

String text = "This cat is cat for cat Sabahs cat Bonus"; String patternString = "cat"; Pattern pattern = Pattern.compile(patternString); String[] split = pattern.split(text);

**Minimize a DFA:** 1. Start with an initial partition II with two groups, F and S -F, the accepting and nonaccepting states of D. 2. Apply the procedure of Fig. 3.64 to construct a new partition IInew· initially, let IInew = II; for ( each group G of II ) {partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group of II; /\* at worst, a state will be in a subgroup by itself \* / replace G in IInew by the set of all subgroups formed; } 3. If IInew = II, let IIfinal = II and continue with step (4). Otherwise, repeat step (2) with IInew in place of II. 4. Choose one state in each group of IIfinal as the representative for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows: (a) The state state of D' is the representative of the group containing the start state of D. (b) The accepting states of D' are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only nonaccepting states, because we started by separating those two classes of states, and the procedure of Fig. 3.64 always forms new groups that are subgroups of previously constructed groups. (c) Let s be the representative of some group G of IIfinal, and let the transition of D from s on input a be to state t. Let r be the rep­ resentative of t's group H. Then in D', there is a transition from s to r on input a. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according