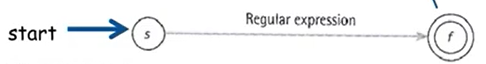
# **Section 11.2.3 Transforming Reg. Exp. into FA**

**Important Facts**

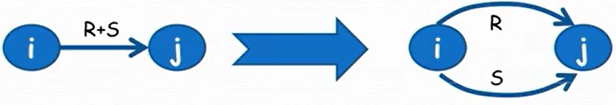
* Regular languages are sets of strings
* Every regular language can be represented by a regular expression
* Every regular expression represents a regular language
* Every regular language can have a DFA and NFA built to recognize it
* If a language is recognized by a DFA and NFA, then it’s regular

## **Algorithm for turning a regular expression into an NFA**

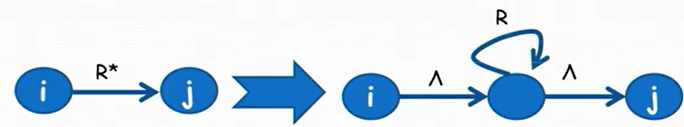
Step 1: Write the regular expression a new kind of machine



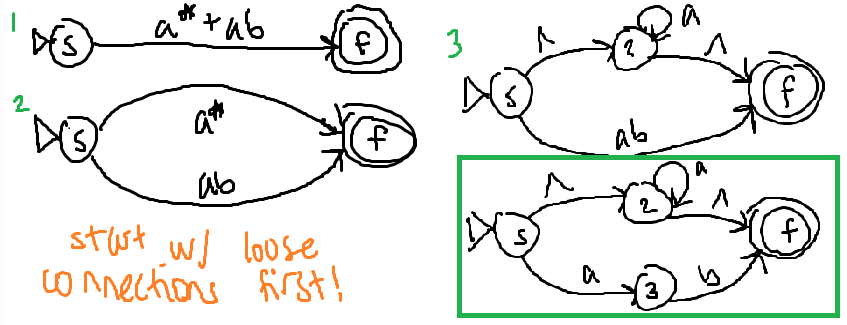
Step 2: Keep applying the following 3 rules until all edges have a letter or ^ (lambda)



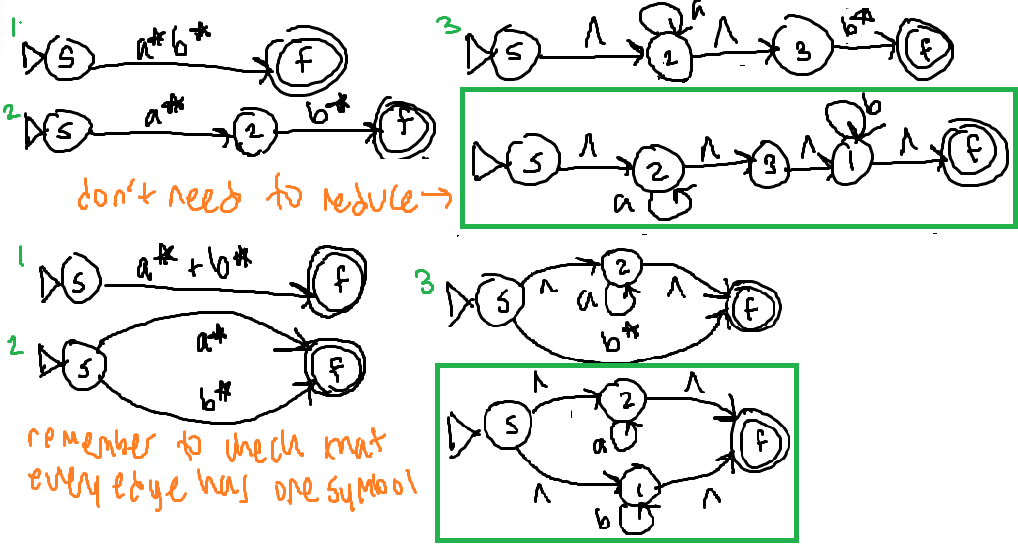




**Example from the book:** Turn a\* + ab into an NFA.

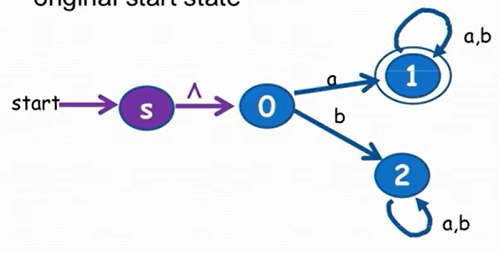


**Problem from the book:** Use the algorithm to transform the following regular expressions into an NFA: a\*b\*, a\* + b\*



## **Algorithm for turning a DFA or NFA into a regular expression (11.2.4)**

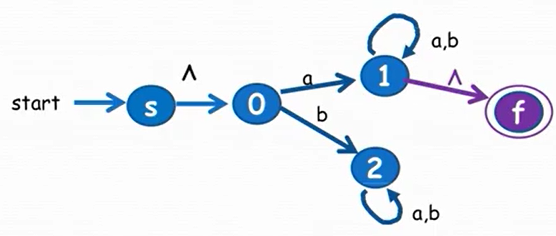
**Example from the book: (pg 757) (shown right):**



**Part 1 <Set Up>**

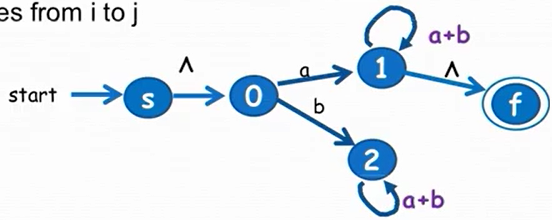
Step 1: Create a new start state *s* & draw a new edge labeled with ^ from *s* to the original start state.

(See right.)

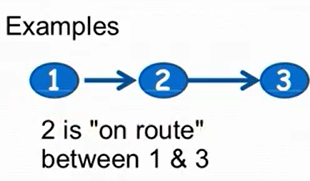


Step 2: Create a final state *f* and draw new edge(s) labelled with ^ from all the accept states to *f*. Eliminate old final states.

(See right.)



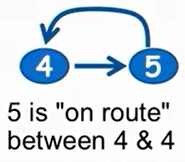
Step 3: If there are two states with more than 1 edge from one to the other, replace all edges with a single edge labelled with the regular expression formed by the sum of the labels of each edge.



**Part 2 <Elimination>**

old(*i*, *j*) = the edge between *i* and *j* on the current machine

Ex. old(0,2) = b; old(0,1) = a; old(1,1) = a + b; old(1,2) = ∅

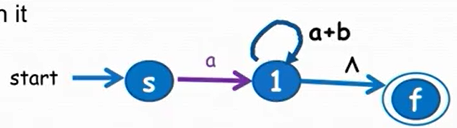
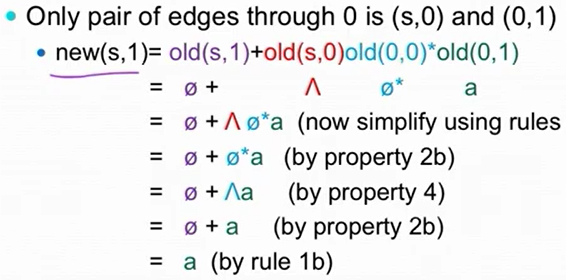
A node *k* is “on route between” nodes *i* and *j* if:

* *i* ≠ *k*
* *j* ≠ *k*
* old(*i*, *k*) ≠ ∅
* old(*k*, *j*) ≠ ∅
* *Note:* It’s OK if *i* == *j*

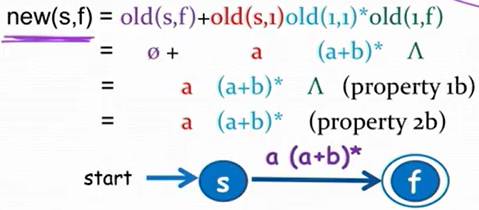
Step 4: You can dump any node that isn’t “on route between” anything. (In the current problem, you can’t get from 2 to the final state anyways, so you can dump it now.)

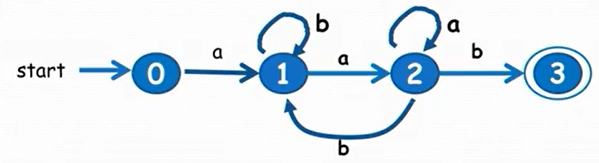
* Then, call the node you want to get rid of *k*
* For every pair *k* is “on route between”, new(*i*, *j*) = old(*i*, *j*) + old(*i*, *k*)old(*k*, *k*)\*old(*k*, *j*)
* This can be done in any order you want

Let’s start with state 0, where state 0 is labelled as *k*, label *s* as *i*, and label 1 as *j*.



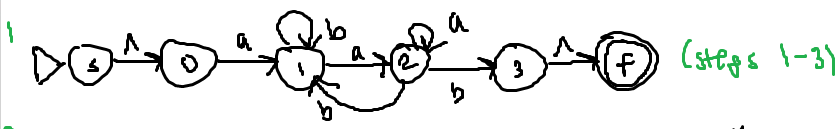
You don’t have to simply but it does make it easier to understand. The new NFA is below it. Next, you can eliminate state 1:

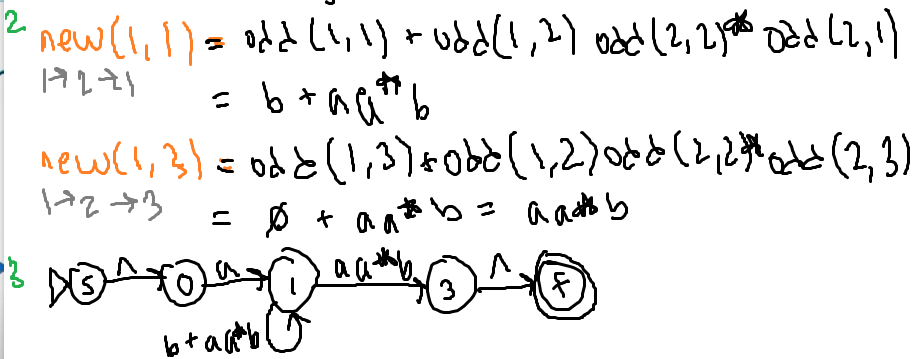


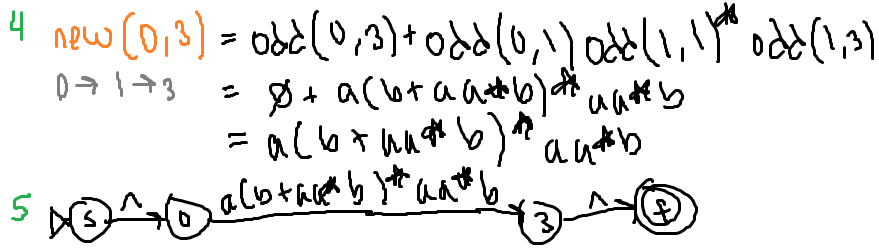


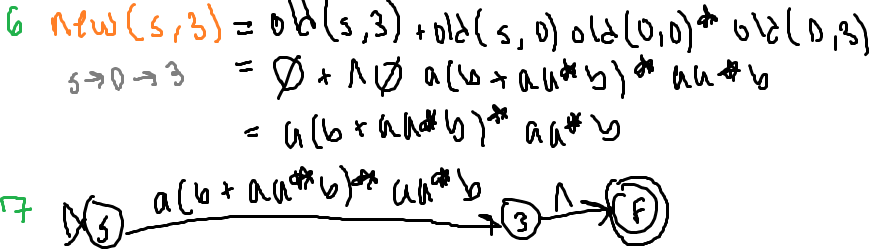
**Problem from the book: (pg 768, #8b):** Use the algorithm to find a regular expression for the language accepted by the right NFA (make sure to delete state 2 before 1 because of the b transition).

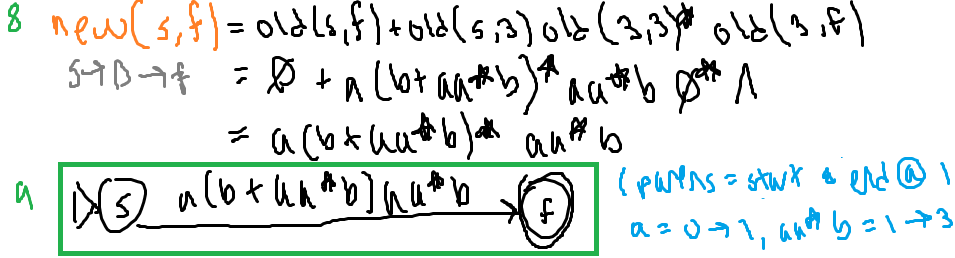
For this one, you need to find new(1,1) and (1,3) because there needs to be ways for things to get back. Route 2 is on route between 1 and 1 as well as 1 and 3. These can be done in the same step before redrawing another one since *k* is 2 for both.

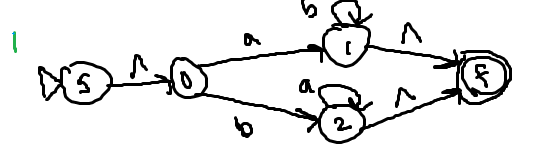


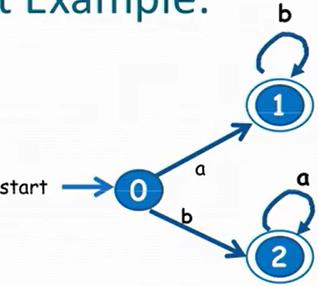


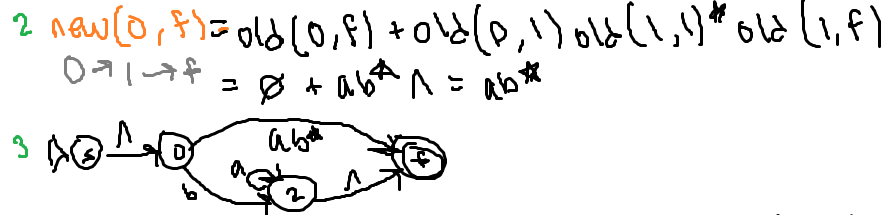


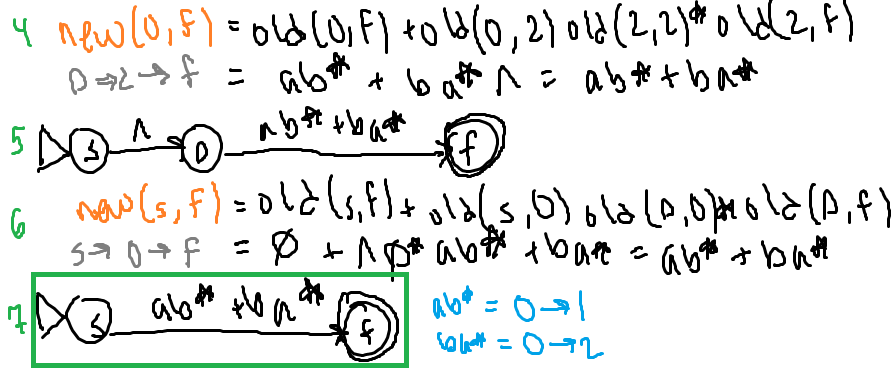






**Another problem (below):** 

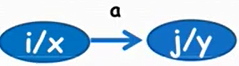




# **Section 11.2.5 Finite Automata as Output Devices**

Interpretation of the Mealy Machine to the right: If you are at state *i* AND you see the letter *a*, then go to state *j* AND print an *x*.

* Notice that “mealy” sounds like “mean”, so think “middle” is where the output is.

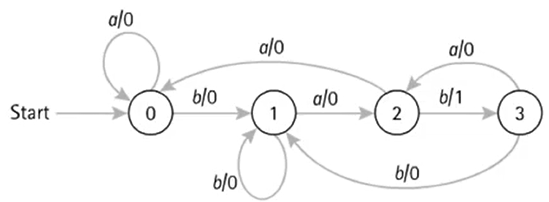


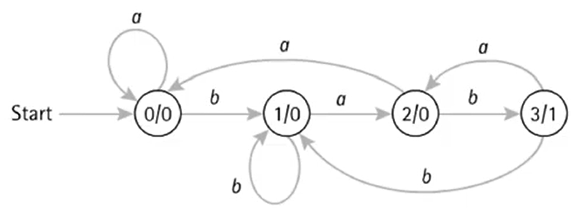
Interpretation of the Moore Machine to the right: Whenever you ENTER the state, you print the letter *x*. If you are at state *i* AND you see the letter *a*, then go to state *j* AND print a y.

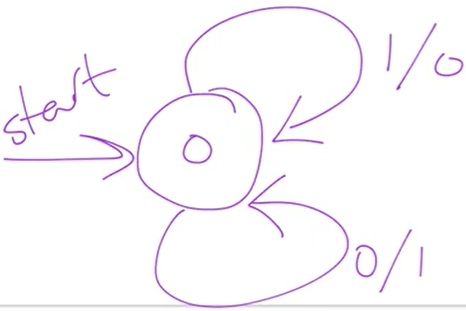
* Notice that “moore” sounds like “more”, so think you get “more” output (one extra).

**Example:** Count the number of bab substrings in a string (overlaps are okay). For example, “abababaababb” has 3.

* The idea is that it checks whether a string has bab’s, so the machine could print out a 1 if the end of a bab substring is recognized. Then at the end, you can count how many 1’s to get the number of bab substrings.
* A Mealy machine to achieve this example is shown below vv

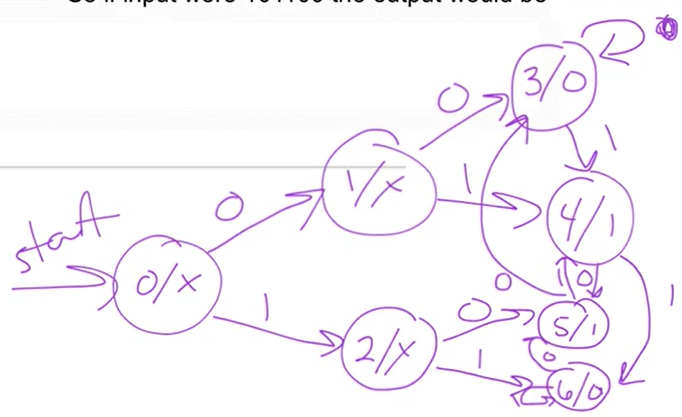


* + State 3: Represents the state where you’ve just seen bab
  + State 2: Represents the state where you’ve just seen ba
  + State 1: Represents the state where you’ve just seen b
  + State 0: Represents the state where you’ve seen aa OR never seen a b (beginning)
* A Moore machine to achieve this example is shown (it prints 1 extra thing, which is the 0 at the start) >>

**Mealy/Moore Machine Examples**

Example: Build a Mealy machine that performs bitwise, NOT on a string of 0’s and 1’s (where the alphabet is {0, 1}). So if the input were 101100, the output would be 010011.

*How this works:* If there is a 1, print a 0 (this print is the output). If there is a 0, print a 1.

Example: Build a Moore machine that performs EXCLUSIVE OR on the last two values seen (where the alphabet = {*x*, 0, 1}). So if the input were 101100, the output would be XX11010.

*How this works:* For exclusive or, if the last two values match, print 0. If those two values don’t match, print 1. *x* is where you haven’t seen two values yet.

Arrive at the start state, and notice you need to print an *x* since you haven’t seen two things yet. You need to print another *x* after that since you’ve only seen one value so far.

*How to come up with it:* Make different states for each possibility of the last two values seen. So there would be 4 possibilities: 00, 01, 10, 11. For each state, you could try thinking of it as what each state has seen. Then draw transitions appropriately after the possibilities for what could happen after. It might be better to think that every state needs to have a transition for (almost) every part of the alphabet.

Make sure to always test out “strings”.

# **Mealy and Moore Machines in JFLAP**

A Mealy machine is similar to a Finite Automaton, with a few key differences:

* It has no final states
* Its ***transitions*** produce output
* It doesn’t accept or reject input; instead, it generates output from input
* It cannot have non-deterministic states (multiple transitions w/ the same input)

When you create a state, you can only set it to initial.

When you create transitions, the input is on the left side while the output is on the right.

Testing works the same. You can go by “Step” to simulate it, or MultipleRun for an output.

You can check for nondeterminism by going on “Test” and “Highlight Nondeterminism”.

As for Moore machines, upon creating a state, it will immediately ask for its output (because the start must have an output). The outputs appear as rectangles on the top right of the states. These have differences from FAs:

* It has no final states
* Its ***states*** produce output
* It doesn’t accept or reject input; instead, it generates output from input
* It cannot have non-deterministic states (multiple transitions w/ the same input)

The transitions are the same (the values are expected or possible inputs).