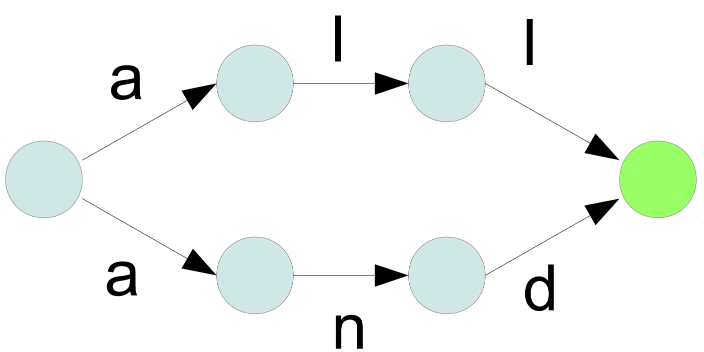
NTNU – Trondheim
Norwegian University of
Science and Technology

NFA to DFA conversion and state minimization

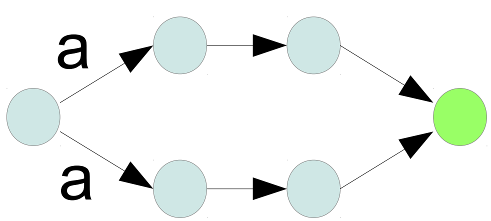
Where we were

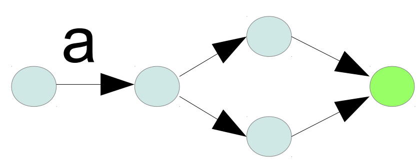
- We have invented a way to turn the regex `(all|and)` into this:
  - (McNaughton, Thompson and Yamada)



So, that doesn't really help right away (dang!)

- We can translate any regex to NFA, but what use is that when our DFA simulation algorithm doesn't work for NFA?
- We'll also have to translate NFA into equivalent DFA
  (*i.e.* there's another thing or two to prove before we're happy)
- Luckily, that's not so hard, it has a lot in common with what we first did when discussing NFA:
  - Find out how far we can take parallel paths before they differ
  - Take those parallel paths and merge them as single states:



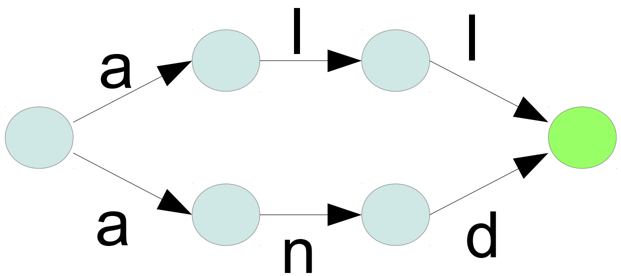


States and sets of states

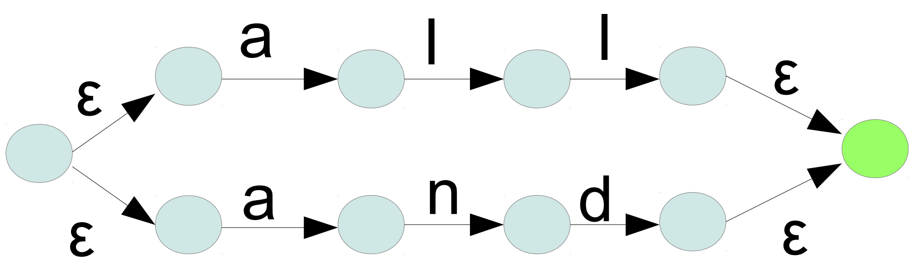
- We'll need to group states together, in order to treat them as one
- Very formally speaking, there is a difference between the state *s* itself and the set {*s*} which has it as the only member
  - I'm going to wave my hands and ignore that difference, because it doesn't add any valuable intuition
  - The exposition in the book cares about the difference, though
- For brevity, let us talk about **S** as if it is a collection of one or more states, and assume that what we say applies to all the states that are included in it.

ε-closure

- Given S in an NFA, its ε-closure is the set of states that can be reached through ε-transitions only
- Once again, this



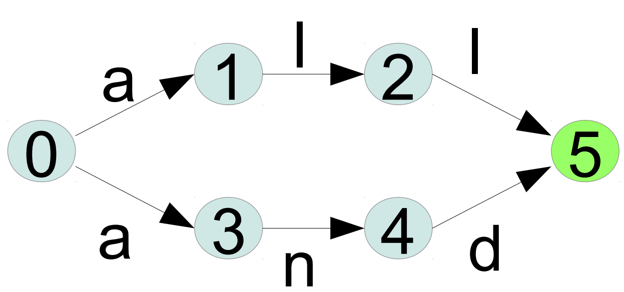
is equivalent to this



NTNU – Trondheim
Norwegian University of
Science and Technology

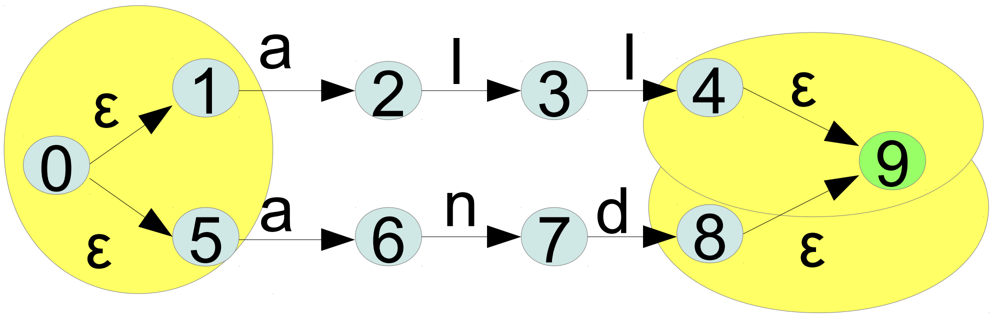
move(S,c)

- move(S,c) is the set of states that you can reach from S when the input character is c
- In DFA-land, this is just the transition table (or function)
  - In the deterministic parts of the automaton below, move(3,n) = {4}, move(2,l) = {5} and so on
- For NFAs, it's a little more interesting
  - move(0,a) = {1,3}



Identifying ε-closures

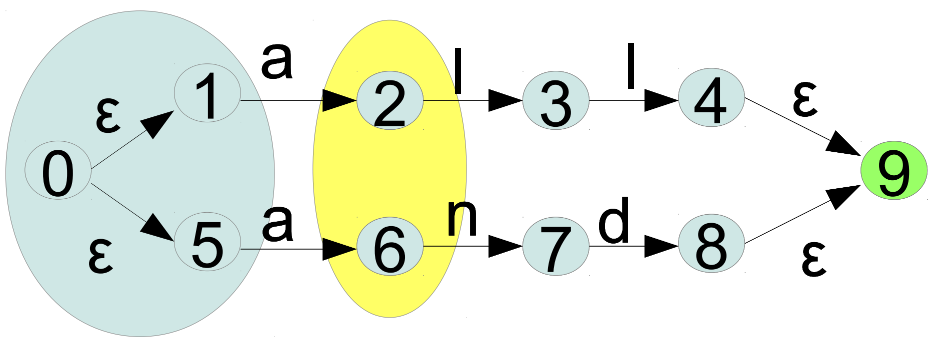
- Numbering the states,



  - ε-closure(0) = {0,1,5}
  - ε-closure(4) = {4,9}
  - ε-closure(8) = {8,9}
- The states in these sets can not be told apart as far as the automaton is concerned

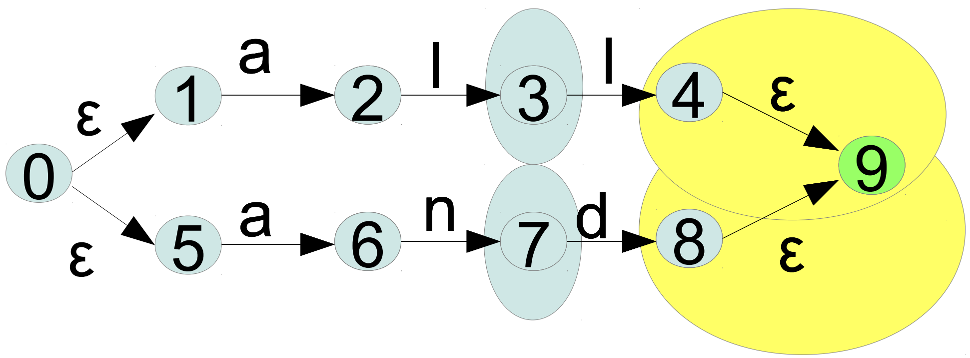
We'll need a group of destinations (let's call it Dtran, for DFA transitions)

- We'll need to collect the transitions that exit the set we want to merge
  - move({0,1,5},a) = {2,6}
  - Dtran[{0,1,5},a] = ε-closure({2,6}) = {2,6}



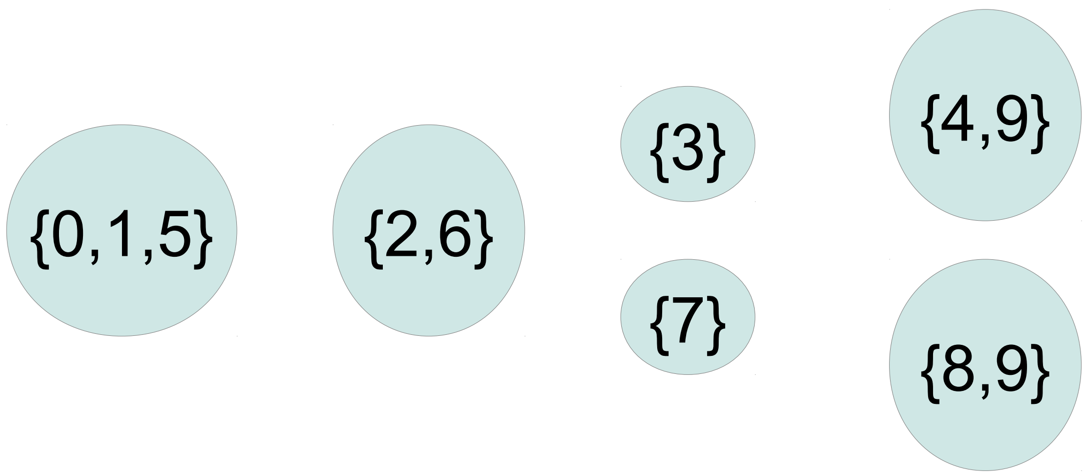
More transitions with multiple destinations

- Dtran is relevant at the other end, too:
  - Dtran[3,l] = ε-closure(move(3,l)) = ε-closure(4) = {4,9}
  - Dtran[7,d] = ε-closure(move(7,d)) = ε-closure(8) = {8,9}



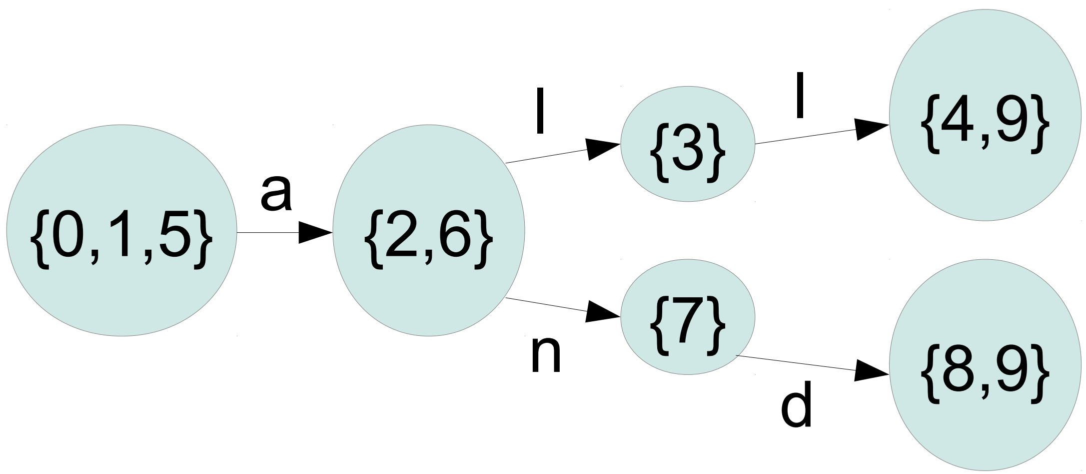
DFA states from indistinguishable sets

- We can now merge the states we have grouped together into new ones that will become our DFA:



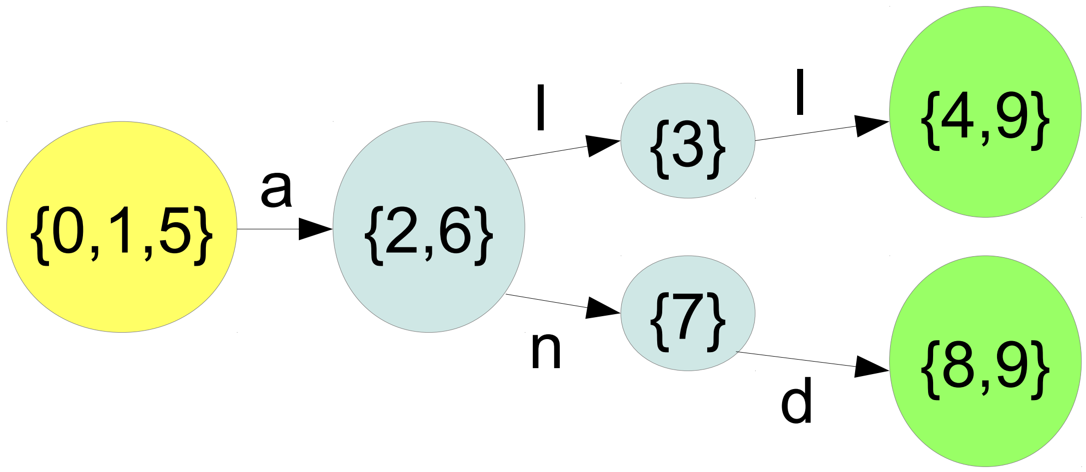
Reintroduce transitions

- Insert the transitions according to Dtran:

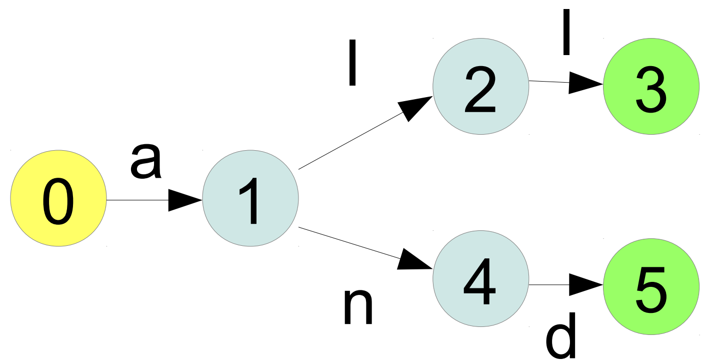


Find the start and end(s)

- If one original state was accepting, any ε-closure that contains it must be accepting, since accept can be reached there without reading any more input



This is a DFA



- It's not quite as economical as our hand-conversion from the beginning
  - There are more states than we need
- It can, however, be constructed automatically
- This method is called *subset construction*

DFA state minimization

- Taking the path regex → NFA → DFA does not *always* introduce useless states
- We have seen that it *can*, though, there's no use for both states 3 and 5 on the previous slide
- They just came out because we were strictly following a set of rules

A matter of space and time

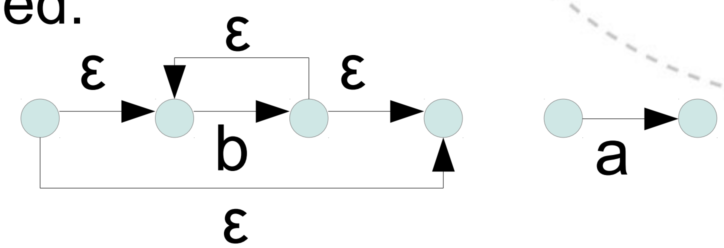
- Minimizing away {3,5} works, but it doesn't illustrate the general procedure very well
- Developing a large DFA with plentiful redundant states doesn't fit nicely into a slide/lecture
- Here's what we can do
  - Take a simple regex which directly gives a minimal DFA
  - Create an equivalent, fluffier DFA by hand and intuition
  - Minimize it, and see that the same result comes out

  (Just mentioning it - if you think that the next example feels a bit contrived, that's because you're perfectly right, it's artificial in order to be small.)

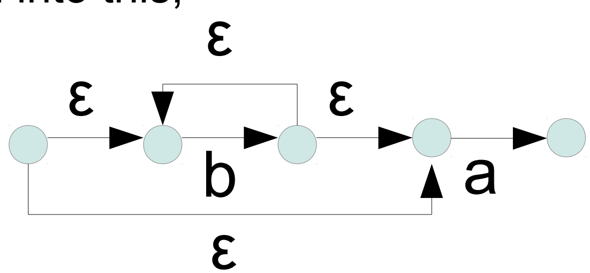
REDO FROM START

- We can quickly take the regex `b*ab*a` through the motions we've already covered:

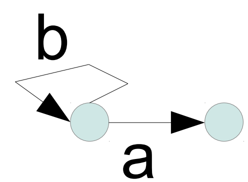
  `b*` and `a` become these,



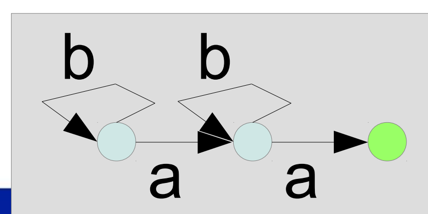
  concatenate them into this,



  merge ε-closures, transitions between subsets,

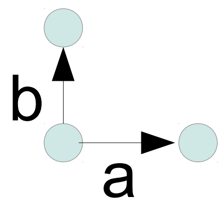


  and concatenate 2 copies:

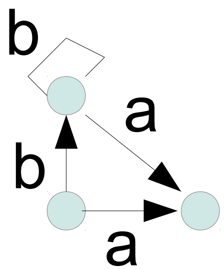


Carelessly, by hand

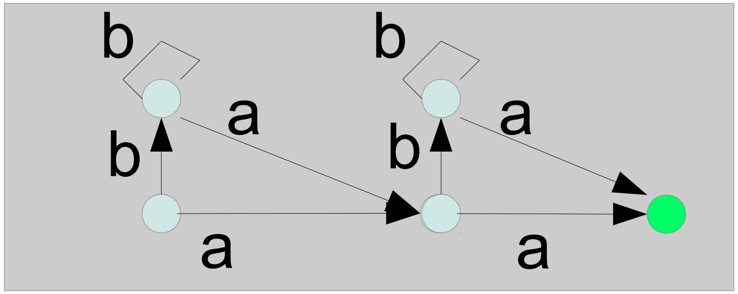
`b*ab*a` must start with either b or a:



Next, there might be any number of b-s, before the mandatory a:



Concatenate 2 of those:

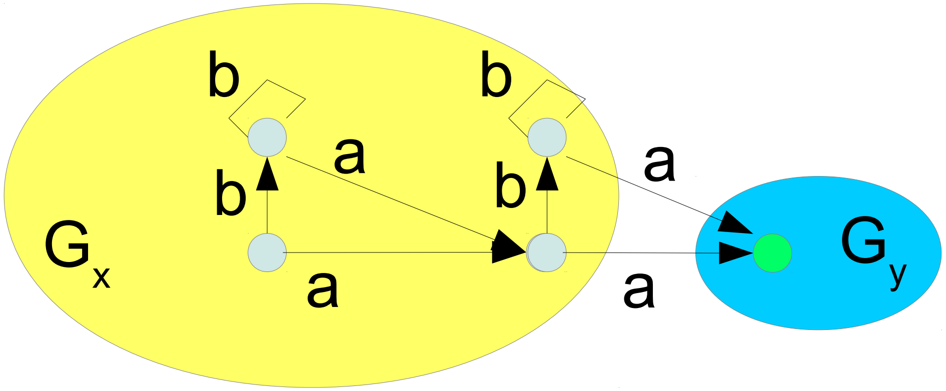


Surely worth minimizing...

NTNU – Trondheim
Norwegian University of Science and Technology

Systematic minimization

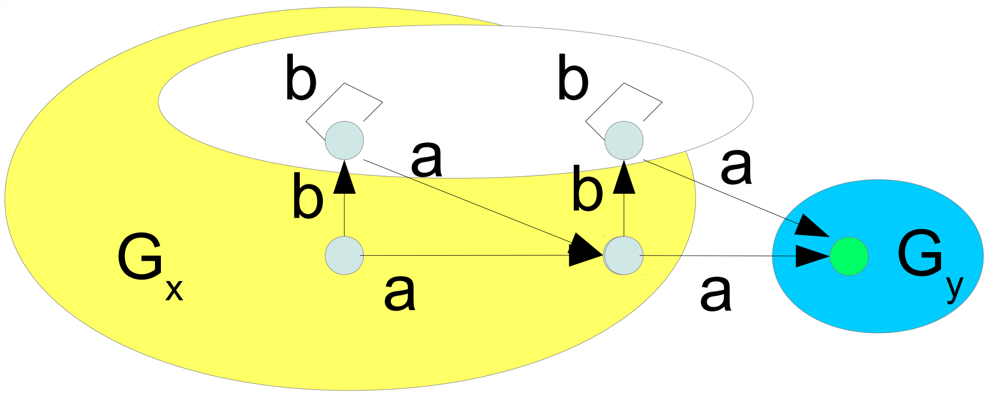
We'll be grouping states together, so start with an initial grouping of non-final and final states



- A pair of states in group G_x are equivalent if and only if their transitions on any given symbol takes them to a state in the same group G_y
- Mind that it's perfectly fine if $G_x = G_y$, the shared destination for a symbol can be the group our pair of states is already in, or a different one

Check a pair for equivalence

This pair is **not** equivalent:

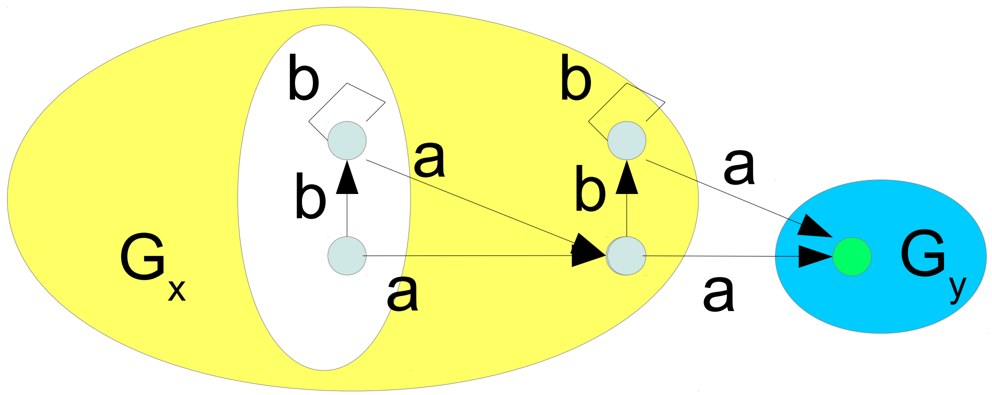


- Both have transitions on b that go from G_x to G_x itself, that's fine
- The leftmost state transitions from G_x to G_x itself on a
- The rightmost transitions from G_x to G_y on a, so we'll need to distinguish between them

NTNU – Trondheim
Norwegian University of
Science and Technology

Check another pair for equivalence

This pair **is** equivalent:

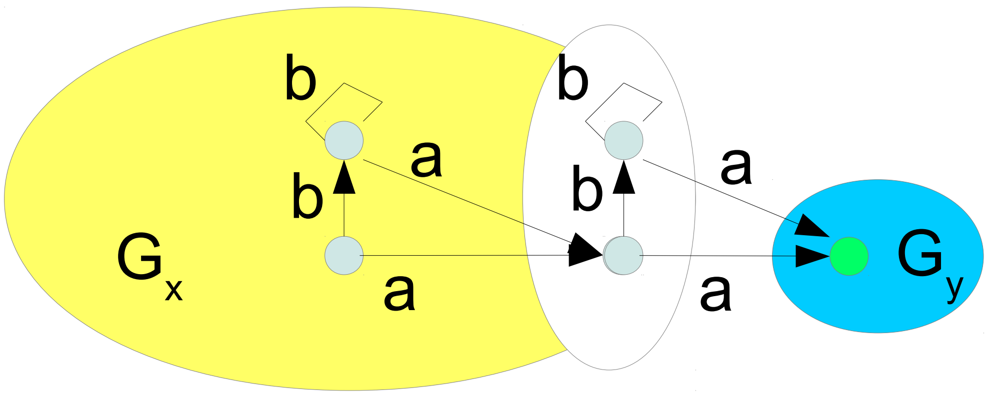


– Both states have transitions on b that go from G_x to G_x itself
– Both states have transitions on a that also go from G_x to G_x itself

Check every pair for equivalence

(at least until you've found one)

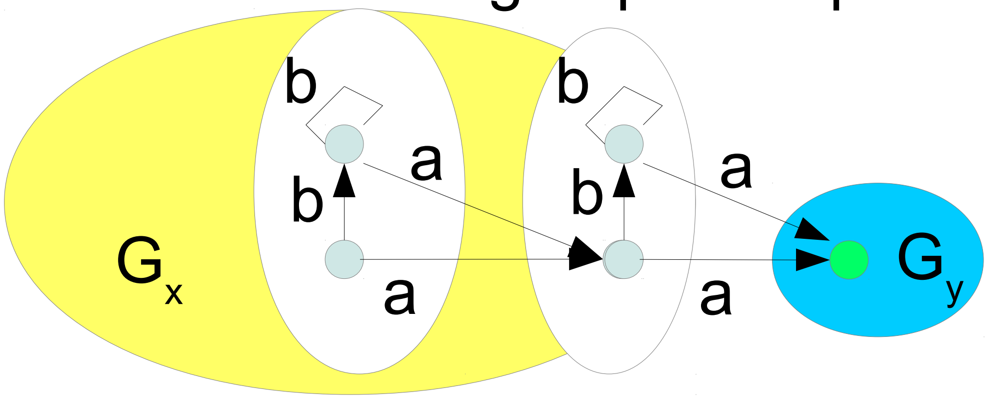
This pair is equivalent as well:



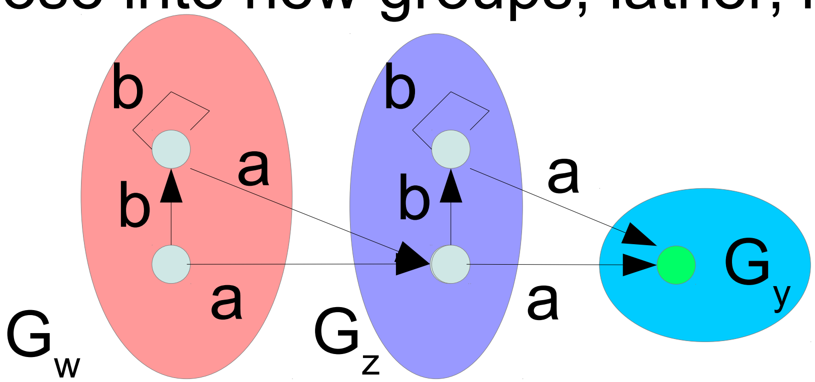
- Both states have transitions on b that go from G_x to G_x itself
- Both states have transitions on a that go from G_x to G_y
- There are three more pairs in G_x, but we can see where this is going without drawing them all…

Divide and conquer

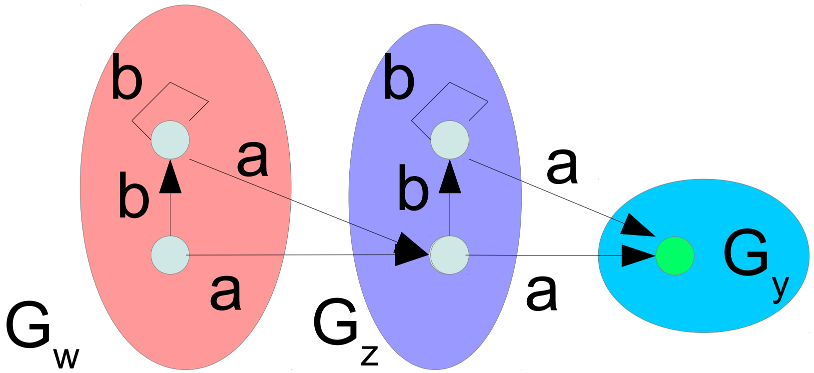
- These are the new groups of equivalent pairs:



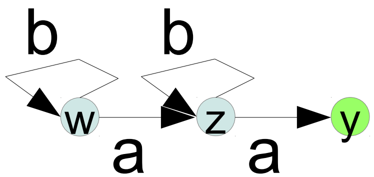
- Split those into new groups, lather, rinse and repeat



In the end



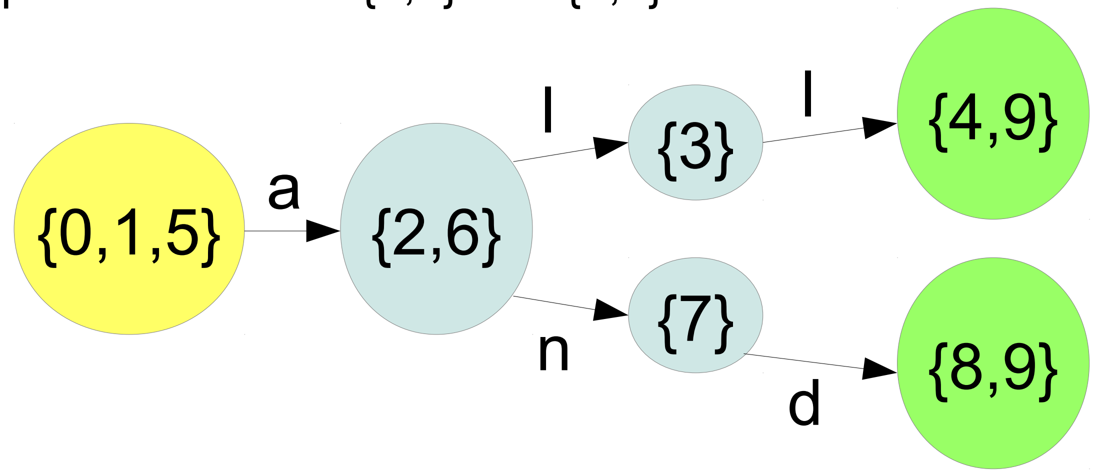
- The pair in G_w is equivalent: a-s take us to G_z, b-s remain in G_w
- The pair in G_z is equivalent: a-s take us to G_y, b-s remain in G_z
- It makes no difference to the rest of the automaton which distinct state within a group we're going to or leaving
- Thus, we might as well make them single states:



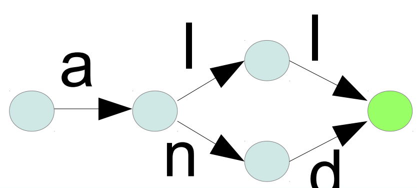
- *Hooray!*

Back where we were

- If you try the same thing with this one, you'll find that the initial grouping into final and non-final states already captures the equivalence of the {4,9} and {8,9} states

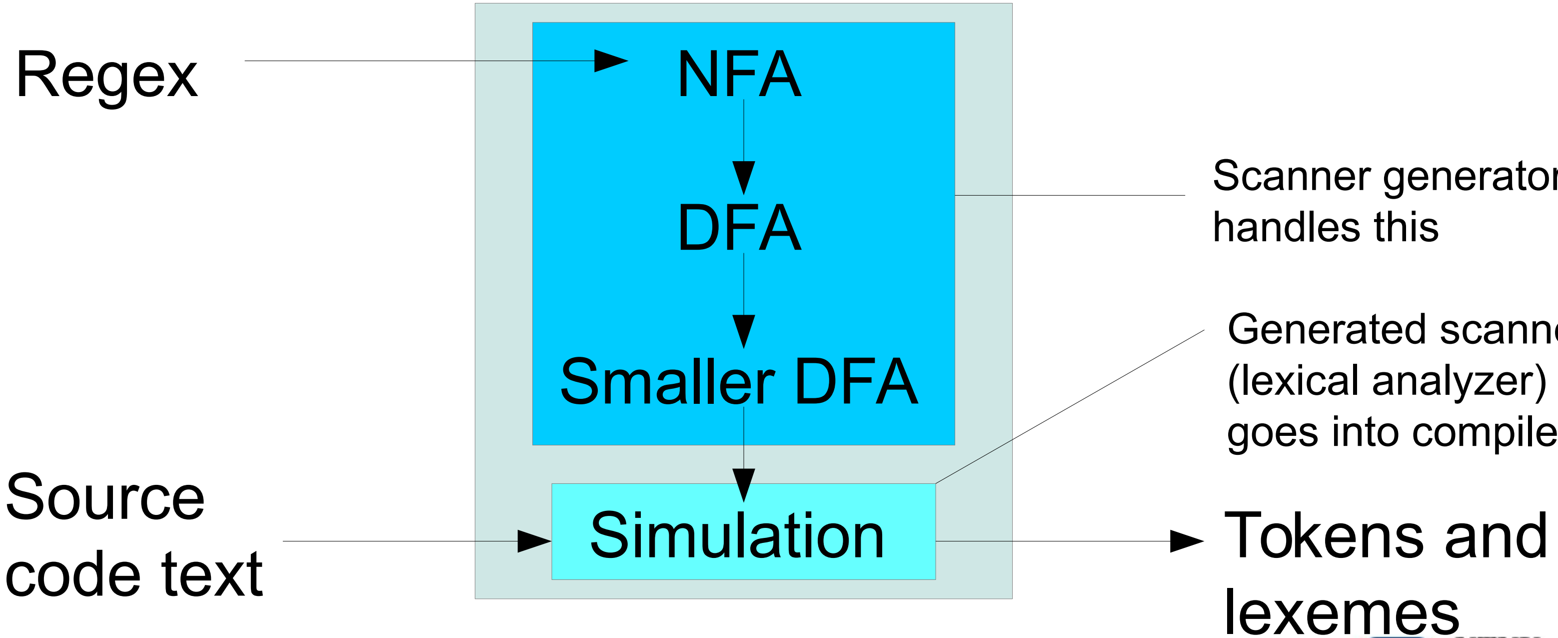


- That creates what we want, but trivial examples are less meaningful



Optimized language acceptors

- We have now seen that this can be done:



The roads not taken

- This is not necessarily *exactly* what happens in a given scanner generator
  - DFA can be made directly from reg.ex.
  - NFA can be simulated on the fly
  - Lookup tables of transitions can be stored more compactly
- My goal is to convince you that there is at least one principled approach to the problem
  - Formal languages and automata theory can be an entire subject
  - Scanning and parsing methods can be one, too
  - We're just borrowing a necessary minimum to Get Things Done™
- I'll round up the loose ends from Chapter 3 next time