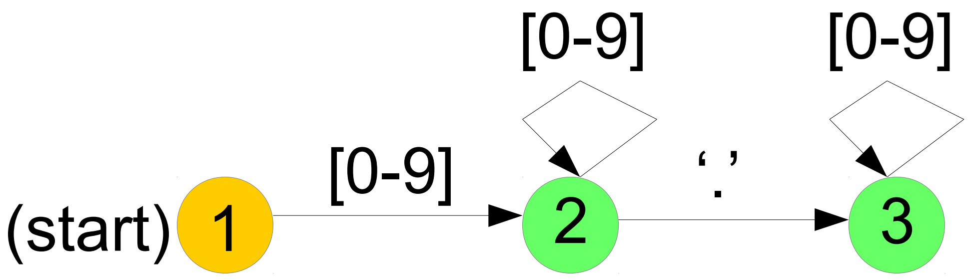
NTNU – Trondheim
Norwegian University of
Science and Technology

# Lexical analysis: Regular Expressions and NFA

# So, we have this DFA

- It can tell you whether or not you have an integer with an optional, fractional part
  - Just point at the first state and the first letter, and follow the arcs



# Common things in lexemes

- Sequences of specific parts
  - These become chains of states in the graph
- Repetition
  - This becomes a loop in the graph
- Alternatives
  - These become different paths that separate and join

# Some notation

- An *alphabet* is any finite set of symbols
  - {0,1} is the alphabet of binary strings
  - [A-Za-z0-9] is the alphabet of alphanumeric strings (English letters)
- Formally speaking, a *language* is a set of valid strings over an alphabet
  - L = {000, 010, 100, 110} is the language of even, positive binary numbers smaller than 8
- A finite automaton *accepts a language*
  - *i.e.* it determines whether or not a string belongs to the language embedded in it by its construction

NTNU – Trondheim
Norwegian University of Science and Technology

# Things we can do with languages

- They can form *unions:*
  - $s \in L_1 \cup L_2$ when $s \in L_1$ or $s \in L_2$
- We can *concatenate* them:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Concatenating a language with itself is a multiplication of sorts (Cartesian product)
  - $LLL = \{ s_1s_2s_3 \mid s_1 \in L \text{ and } s_2 \in L \text{ and } s_3 \in L\} = L^3$
- We can find *closures*
  - $L^* = \cup_{i=0,1,2,\ldots} L^i$ (Kleene closure) ← sequences of 0 or more strings from L
  - $L^+ = \cup_{i=1,2,\ldots} L^i$ (Positive closure) ← sequences of 1 or more strings from L

NTNU – Trondheim
Norwegian University of
Science and Technology

# *Regular expressions*

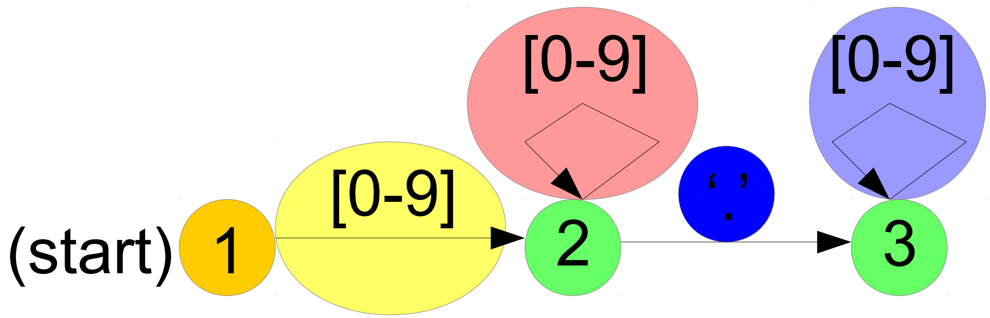
(“regex”, among friends)

- We denote the empty string as *ε* (epsilon)
- The alphabet of symbols is denoted Σ (sigma)
- **Basis**
  - *ε* is a regular expression, L(*ε*) is the language with only ε in it
  - If *a* is in Σ, then *a* is also a regular expression (symbols can simply be written into the expression), L(*a*) is the language with only *a* in it
- **Induction**
  - If $r_1$ and $r_2$ are regular expressions, then $\mathbf{r_1 \mid r_2}$ is a reg.ex. for $L(r_1) \cup L(r_2)$
    (selection, *i.e.* “either $r_1$ or $r_2$”)
  - If $r_1$ and $r_2$ are regular expressions, then $\mathbf{r_1 r_2}$ is a reg.ex. for $L(r_1)L(r_2)$
    (concatenation)
  - If r is a regular expression, then r* denotes L(r)*
    (Kleene closure)
  - (r) is a regular expression denoting L(r)
    (We can add parentheses)

NTNU – Trondheim
Norwegian University of Science and Technology

# DFA and regular expressions
(superficially)

- We already noted that this thing recognizes a language because of how it's constructed:




- There's a corresponding regular expression:

    [0-9] [0-9]* ( . [0-9]* )?

    Optional, because state 2 accepts

NTNU – Trondheim
Norwegian University of
Science and Technology

# Now there are 3 views

- Graphs, for sorting things out
- Tables, for writing programs that do what the graph does
- Regular expressions, for generating them automatically

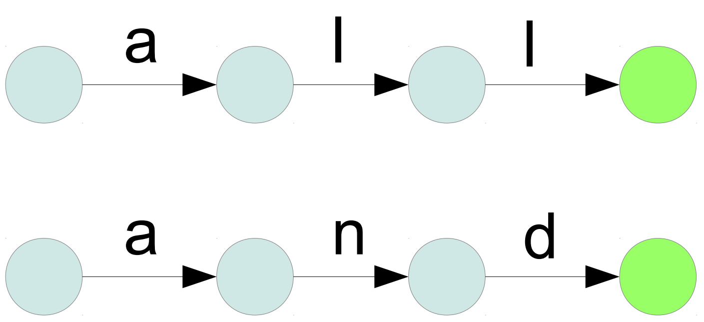
# Regular languages

- All our representations show the same thing
  - We haven't shown how to construct either one from the other, but maybe you can see it still.
- The family of all the languages that can be recognized by reg.ex. / automata are called the *regular languages*
- They're a pretty powerful programming tool on their own, but they don't cover *everything*

  (more on that later)

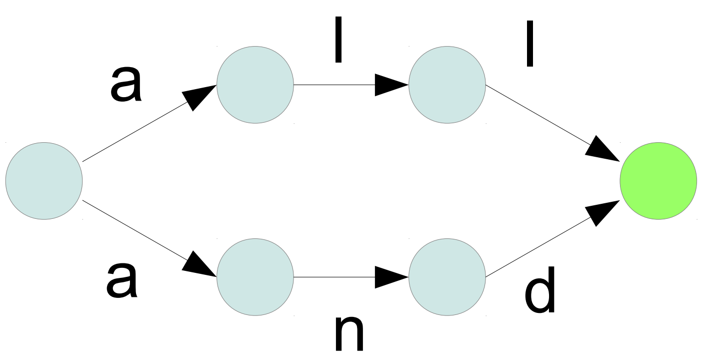
# Combining automata

- Suppose we want a language which includes both of the words {"all", "and"}
- Separately, these make simple DFA:



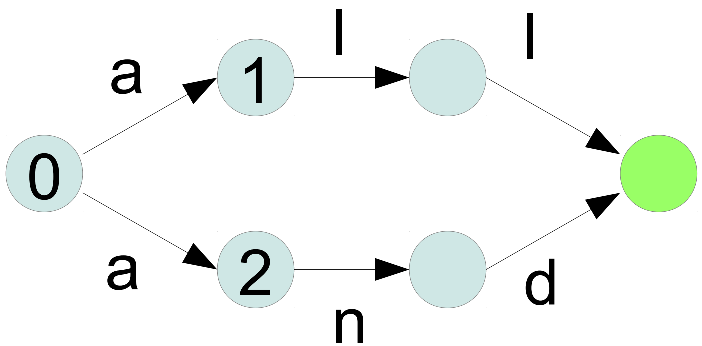
# Putting them together

- The easiest way we could combine them into an automaton which recognizes both, is to just glue their start and end states together:



# This is *slightly* problematic

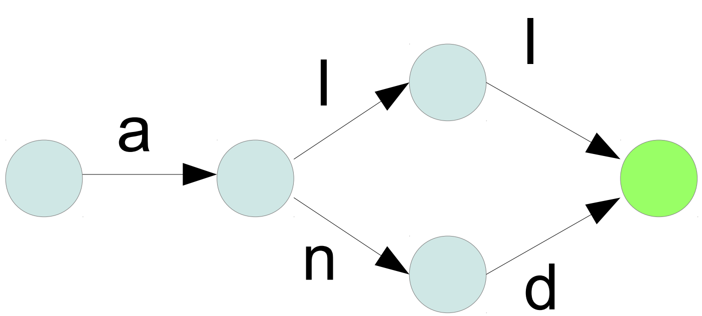
- The simulation algorithm from last time doesn't work that way:
  - Starting from state 0 and reading 'a', the next state can be either 1 or 2
  - If we went from 0 to 1 on an 'a' and next see an 'n', we should have gone with state 2 instead
  - If we see an 'a' in state 0, the only safe bet against having to back-track is to go to states 1 *and* 2 at the same time...




NTNU – Trondheim
Norwegian University of Science and Technology

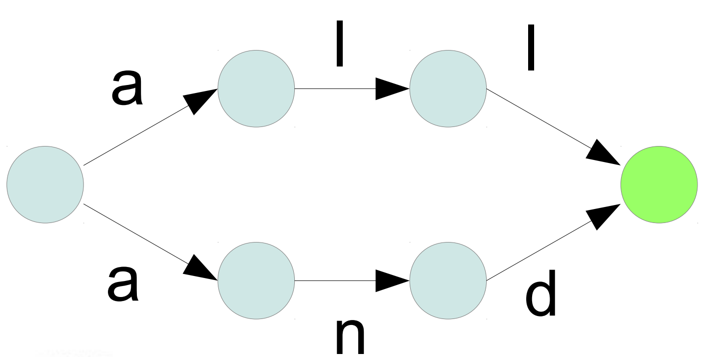
# The obvious solution

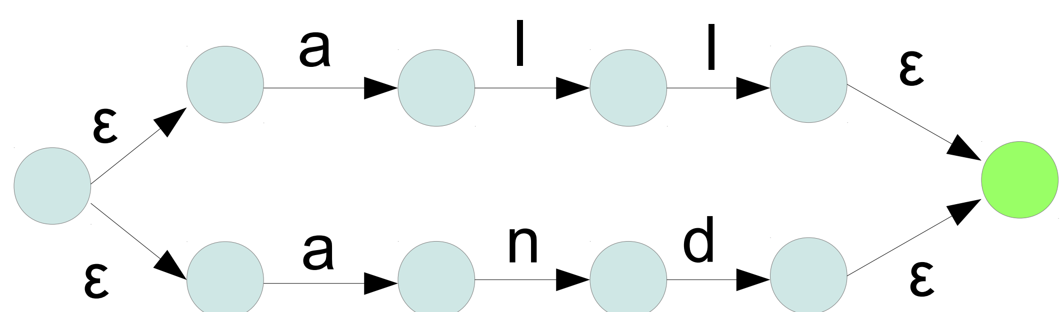
- Join states 1 and 2, thus postponing the choice of paths until it matters:
- Now the simple algorithm works again (*yay!*)
- ...but we had to analyze what our two words have in common (*how general is that?*)



# Non-deterministic Finite Automata

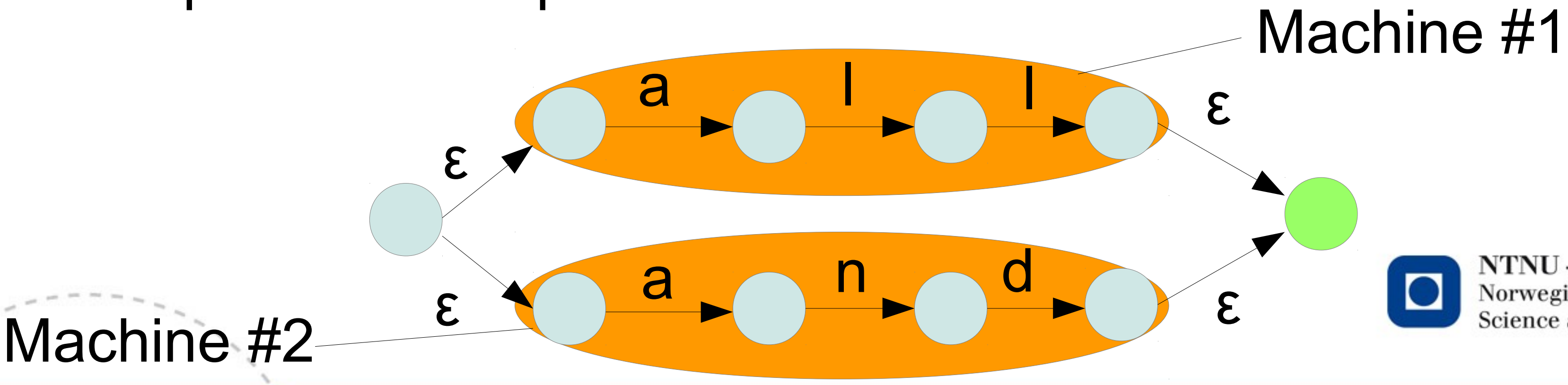
- One way to write an NFA is to admit multiple transitions on the same character
- Another is to admit transitions on the empty string, which we already denoted as “ε” (epsilon)
- These are equivalent notations for the same idea:



# Relation to regular expressions

- NFA are easy to make from regular expressions
- The pair of words we already looked at can be recognized as the regex `( all | and )`
  - (equivalently, `a( ll | nd )` for the deterministic variant, but never mind for the moment)
- We can easily recognize the sub-automata from each part of the expression:



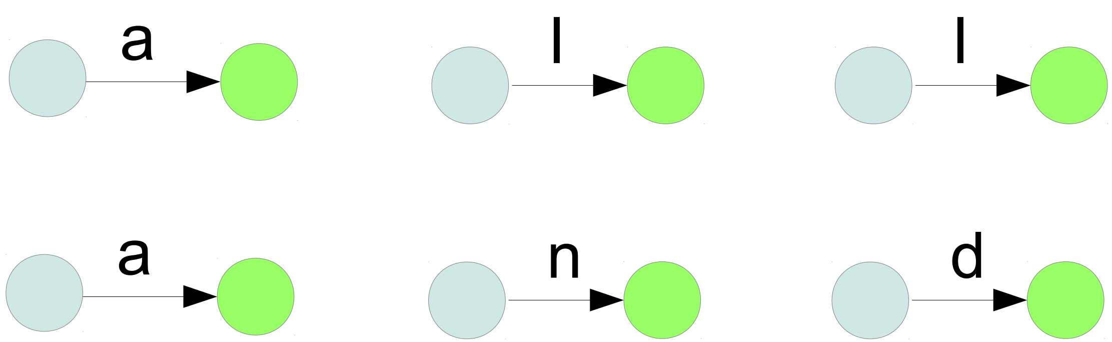

NTNU – Trondheim
Norwegian University of Science and Technology

# What can a regex contain?

- Let's revisit the definition:
  1) a character stands for itself *(or epsilon, but that's invisible)*
  2) concatenation $R_1 R_2$
  3) selection $R_1 \mid R_2$
  4) grouping $(R_1)$
  5) Kleene closure $R_1^*$
- We can show how to construct NFA for each of these, all we need to know is that $R_1$, $R_2$ are regular expressions
- Notice that a DFA is also an NFA
  - It just happens to contain zero ε-transitions
  - More properly put, DFA are a subset of NFA

# 1) A character

- Single characters (and epsilons) in a regex become transitions between two states in an NFA
- Working from `( all | and )`, that gives us

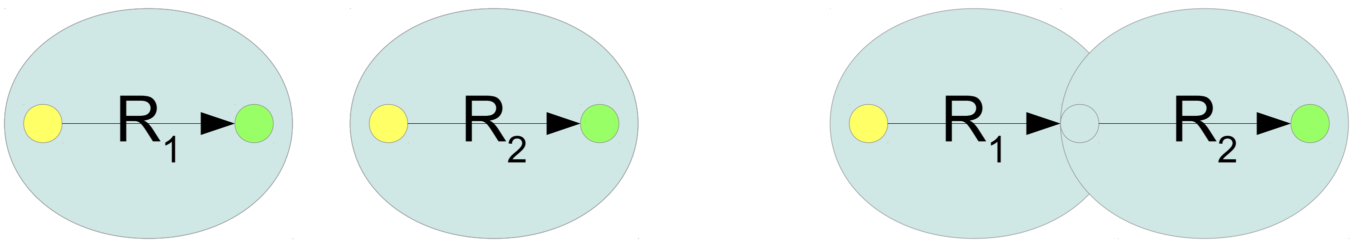



Now we have a bunch of tiny Rs to combine

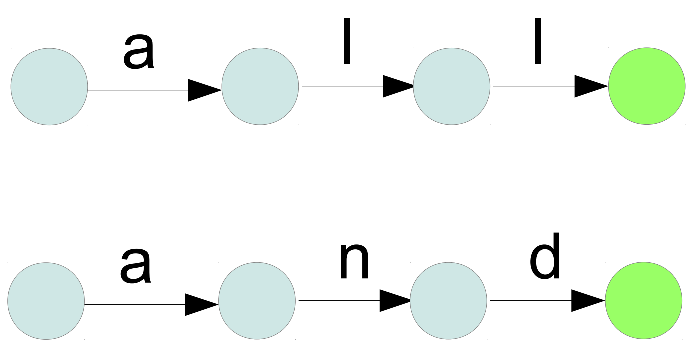
NTNU – Trondheim
Norwegian University of Science and Technology

# 2) Concatenation

- Where $R_1R_2$ are concatenated, join the accepting state of $R_1$ with the start state of $R_2$:



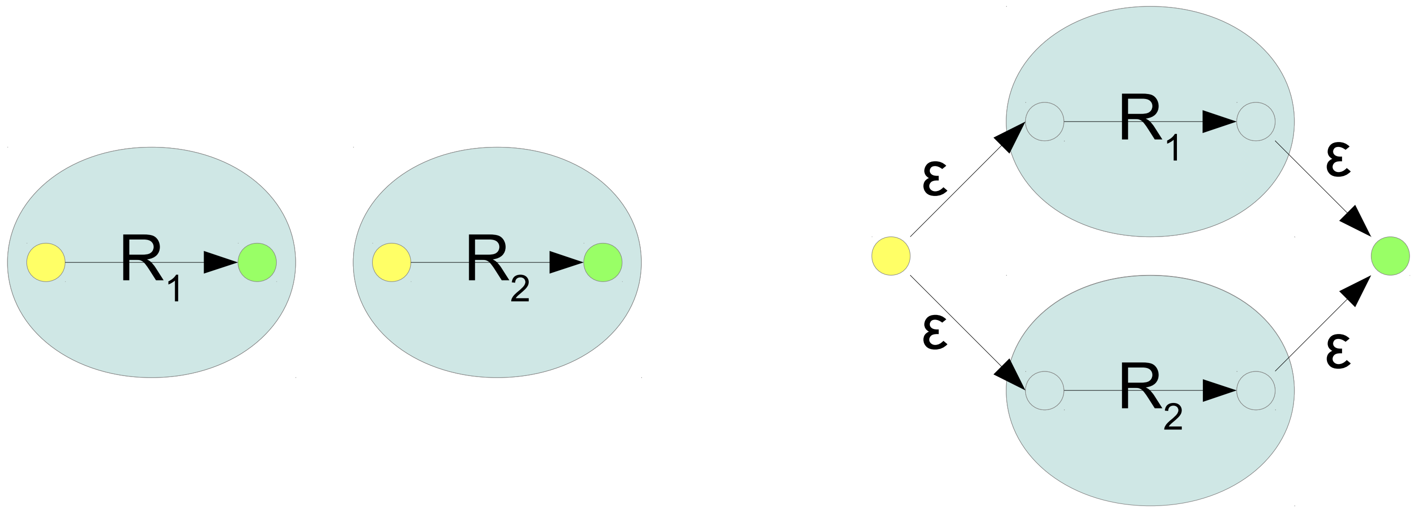

- In our example:




NTNU – Trondheim
Norwegian University of Science and Technology

# 3) Selection

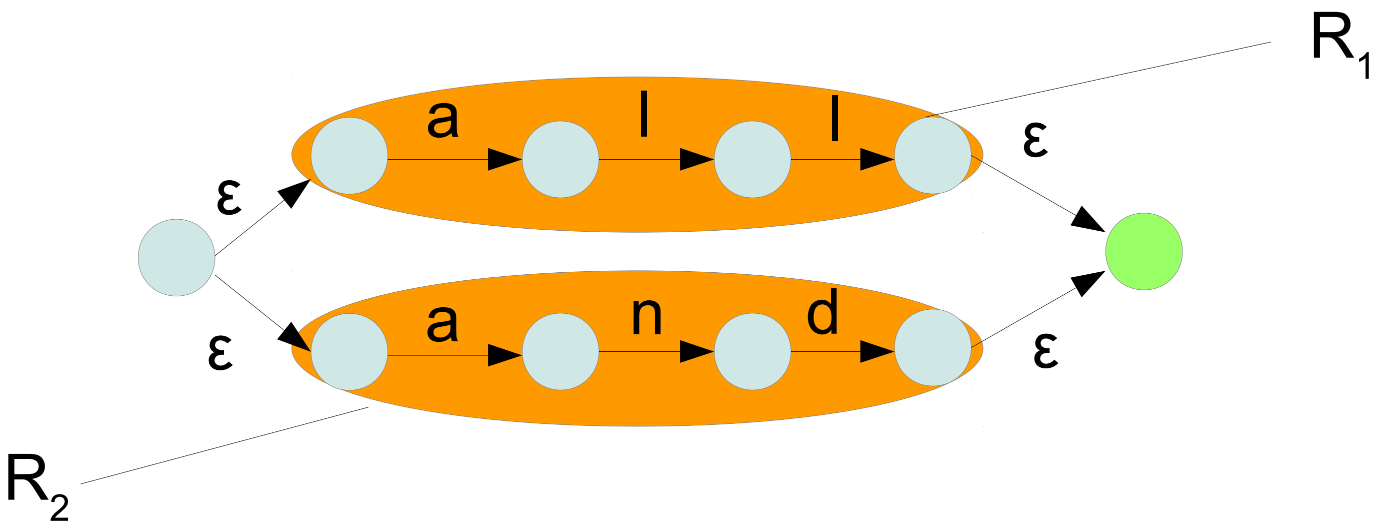
- Introduce new start+accept states, attach them using ε-transitions (so as not to change the language):




NTNU – Trondheim
Norwegian University of
Science and Technology

# (That completes the example)

- It's exactly what we did before:




NTNU – Trondheim
Norwegian University of
Science and Technology

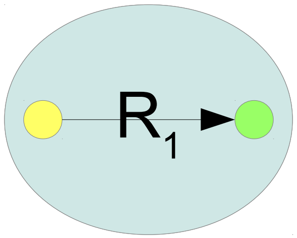
# 4) Grouping

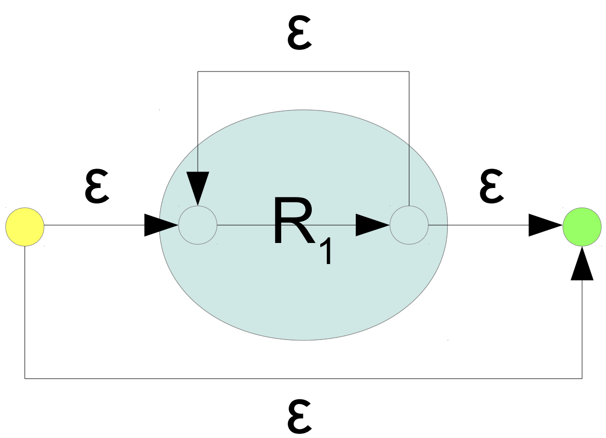
- Parentheses just delimit which parts of an expression to treat as a (sub-)automaton, they appear in the form of its structure, but not as nodes or edges
- *cf.* how the automaton for `(all|and)` will be exactly the same as that for `((a)(l)(l))|((a)(n)(d))`

NTNU – Trondheim
Norwegian University of Science and Technology

# 5) Kleene closure

- $R_1$* means zero or more concatenations of $R_1$
- Introduce new start/accept states, and ε-transitions to
  - Accept one trip through $R_1$
  - Loop back to its beginning, to accept any number of trips
  - Bypass it entirely, to accept zero trips



# Q.E.D.

- We have now proven that an NFA can be constructed from <u>any</u> regular expression
  - None of these maneuvers depend on what the expressions contain
- It's the *McNaughton-Thompson-Yamada algorithm*

  (Bear with me if I accidentally call it "Thompson's construction", it's the same thing, but previous editions of the Dragon used to short-change McNaughton and Yamada)
- But wait… what about the positive closure, $R_1^+$?
  - It can be made from concatenation and Kleene closure, try it yourself
  - It's handy to have as notation, but not necessary to prove what we wanted here

# One lucid moment

- We’ve talked about *closures*
  - They are the outcome of repeating a rule until the result stops changing (possibly never)
- We’ve taken a notation and attached general rules to all its elements, one at a time
  - By induction, this guarantees that we cover all their combinations
  - That is the trick of a “syntax directed definition”
- Hang on to these ideas
  - They will appear often in what lies ahead of us