Chapter Seven:
Regular Expressions

The first time a young student sees the mathematical constant π, it looks like just one more school artifact: one more arbitrary symbol whose definition to memorize for the next test. Later, if he or she persists, this perception changes. In many branches of mathematics and with many practical applications, π keeps on turning up. "There it is again!" says the student, thus joining the ranks of mathematicians for whom mathematics seems less like an artifact invented and more like a natural phenomenon discovered.

So it is with regular languages. We have seen that DFAs and NFAs have equal definitional power. It turns out that regular expressions also have exactly that same definitional power: they can be used to define all the regular languages, and only the regular languages. There it is again!

Outline

- 7.1 Regular Expressions, Formally Defined
- 7.2 Regular Expression Examples
- 7.3 For Every Regular Expression, a Regular Language
- 7.4 Regular Expressions and Structural Induction
- 7.5 For Every Regular Language, a Regular Expression

Concatenation of Languages

- The concatenation of two languages L_1 and L_2 is $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- The set of all strings that can be constructed by concatenating a string from the first language with a string from the second
- For example, if $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$ then $L_1L_2 = \{ac, ad, bc, bd\}$

Kleene Closure of a Language

- The Kleene closure of a language L is $L^* = \{x_1 x_2 \ldots x_n \mid n \geq 0, \text{ with all } x_i \in L\}$
- The set of strings that can be formed by concatenating any number of strings, each of which is an element of L
- Not the same as $\{x^n \mid n \geq 0 \text{ and } x \in L\}$
- In L^*, each x_i may be a different element of L
- For example, $\{ab, cd\}^* = \{\varepsilon, ab, cd, abab, abcd, cdab, cdcd, ababab, \ldots\}$
- For all L, $\varepsilon \in L^*$
- For all L containing at least one string other than ε, L^* is infinite

Regular Expressions

- A regular expression is a string *r* that denotes a language *L*(*r*) over some alphabet Σ
- Regular expressions make special use of the symbols ε, $\varnothing$, +, *, and parentheses
- We will assume that these special symbols are not included in Σ
- There are six kinds of regular expressions…

The Six Regular Expressions

- The six kinds of regular expressions, and the languages they denote, are:
  - Three kinds of *atomic* regular expressions:
    - Any symbol $a \in \Sigma$, with $L(a) = \{a\}$
    - The special symbol ε, with $L(\varepsilon) = \{\varepsilon\}$
    - The special symbol $\varnothing$, with $L(\varnothing) = \{\}$
  - Three kinds of *compound* regular expressions built from smaller regular expressions, here called r, r_1, and r_2:
    - $(r_1 + r_2)$, with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
    - $(r_1 r_2)$, with $L(r_1 r_2) = L(r_1)L(r_2)$
    - $(r)^*$, with $L((r)^*) = (L(r))^*$
- The parentheses may be omitted, in which case * has highest precedence and + has lowest

Other Uses of the Name

- These are classical regular expressions
- Many modern programs use text patterns also called *regular expressions*:
  - Tools like awk, sed and grep
  - Languages like Perl, Python, Ruby, and PHP
  - Language libraries like those for Java and the .NET languages
- All slightly different from ours and each other
- More about them in a later chapter

Outline

- 7.1 Regular Expressions, Formally Defined
- 7.2 Regular Expression Examples
- 7.3 For Every Regular Expression, a Regular Language
- 7.4 Regular Expressions and Structural Induction
- 7.5 For Every Regular Language, a Regular Expression

ab

- Denotes the language {*ab*}
- Our formal definition permits this because
  - *a* is an atomic regular expression denoting {*a*}
  - *b* is an atomic regular expression denoting {*b*}
  - Their concatenation (*ab*) is a compound
  - Unnecessary parentheses can be omitted
- Thus any string *x* in Σ^* can be used by itself as a regular expression, denoting {*x*}

ab+*c*

- Denotes the language {*ab*,*c*}
- We omitted parentheses from the fully parenthesized form ((*ab*)+*c*)
- The inner pair is unnecessary because + has lower precedence than concatenation
- Thus any finite language can be defined using a regular expression
- Just list the strings, separated by +

*ba**

- Denotes the language {ba^n}: the set of strings consisting of *b* followed by zero or more *a*s
- Not the same as (*ba*)*, which denotes {$(ba)^n$}
- * has higher precedence than concatenation
- The Kleene star is the only way to define an infinite language using regular expressions

$(a+b)^*$

- Denotes $\{a,b\}^*$: the whole language of strings over the alphabet $\{a,b\}$
- The parentheses are necessary here, because * has higher precedence than +
- $a+b^*$ denotes $\{a\} \cup \{b\}^*$
- Reminder: not "zero or more copies…"
- That would be a^*+b^*, which denotes $\{a\}^* \cup \{b\}^*$

ab+ε

- Denotes the language {*ab*,ε}
- Occasionally, we need to use the atomic regular expression ε to include ε in the language
- But it's not needed in (*a*+*b*)*+ε, because ε is already part of every Kleene star

∅

- Denotes {}
- There is no other way to denote the empty set with regular expressions
- That's all you should ever use ∅ for
- It is not useful in compounds:
  - $L(r\emptyset) = L(\emptyset r) = \{\}$
  - $L(r+\emptyset) = L(\emptyset+r) = L(r)$
  - $L(\emptyset^*) = \{\varepsilon\}$

More Examples

- $(a+b)(c+d)$
  - Denotes $\{ac, ad, bc, bd\}$
- $(abc)^*$
  - Denotes $\{(abc)^n\} = \{\varepsilon, abc, abcabc, \ldots\}$
- a^*b^*
  - Denotes $\{a^n b^m\} = \{xy \mid x \in \{a\}^* \text{ and } y \in \{b\}^*\}$

More Examples

- $(a+b)^*aa(a+b)^*$
  - Denotes $\{x \in \{a,b\}^* \mid x$ contains at least 2 consecutive as$\}$
- $(a+b)^*a(a+b)^*a(a+b)^*$
  - Denotes $\{x \in \{a,b\}^* \mid x$ contains at least 2 as$\}$
- $(a^*b^*)^*$
  - Denotes $\{a,b\}^*$, same as the simpler $(a+b)^*$
  - Because $L(a^*b^*)$ contains both a and b, and that's enough: we already have $L((a+b)^*) = \{a,b\}^*$
  - In general, whenever $\Sigma \subseteq L(r)$, then $L((r)^*) = \Sigma^*$

Outline

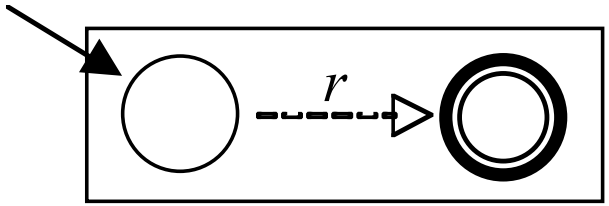
- 7.1 Regular Expressions, Formally Defined
- 7.2 Regular Expression Examples
- 7.3 For Every Regular Expression, a Regular Language
- 7.4 Regular Expressions and Structural Induction
- 7.5 For Every Regular Language, a Regular Expression

Regular Expression to NFA

- Goal: to show that every regular expression defines a regular language
- Approach: give a way to convert any regular expression to an NFA for the same language
- Advantage: large NFAs can be composed from smaller ones using ε-transitions

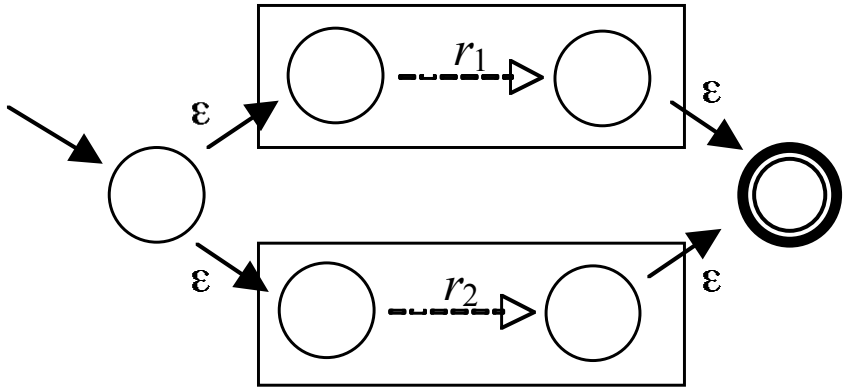
Standard Form

- To make them easier to compose, our NFAs will all have the same standard form:
  - Exactly one accepting state, not the start state
- That is, for any regular expression *r*, we will show how to construct an NFA *N* with *L*(*N*) = *L*(*r*), pictured like this:



Composing Example

- That form makes composition easy
- For example, given NFAs for $L(r_1)$ and $L(r_2)$, we can easily construct one for $L(r_1+r_2)$:



- This new NFA still has our special form

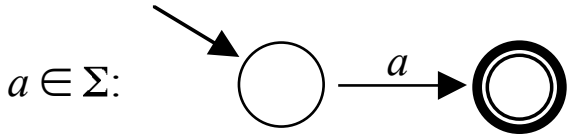
Lemma 7.3

If *r* is any regular expression, there is some NFA *N* that has a single accepting state, not the same as the start state, with $L(N) = L(r)$.

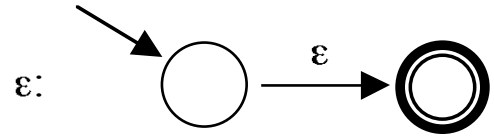
- Proof sketch:
  - There are six kinds of regular expressions
  - We will show how to build a suitable NFA for each kind

Proof Sketch: Atomic Expressions

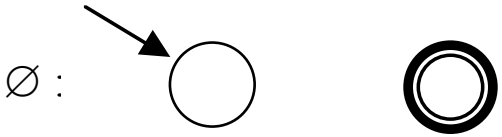
- There are three kinds of atomic regular expressions
  - Any symbol $a \in \Sigma$, with $L(a) = \{a\}$



  - The special symbol ε, with $L(\varepsilon) = \{\varepsilon\}$



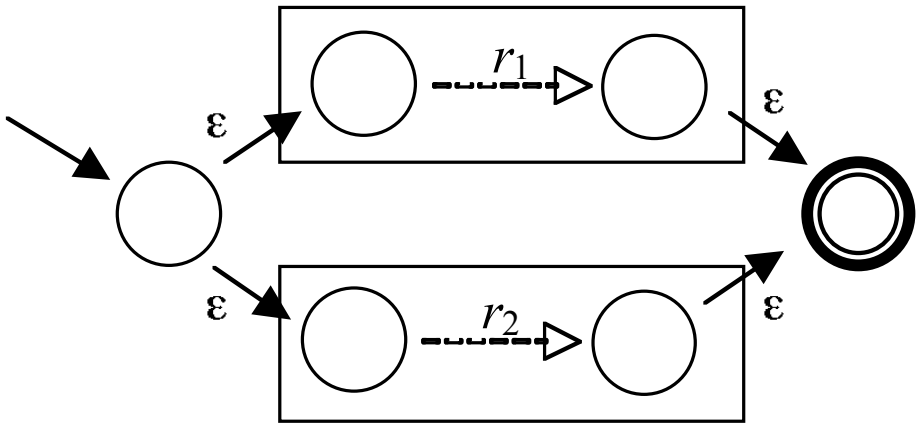
  - The special symbol $\varnothing$, with $L(\varnothing) = \{\}$



Proof: Compound Expressions

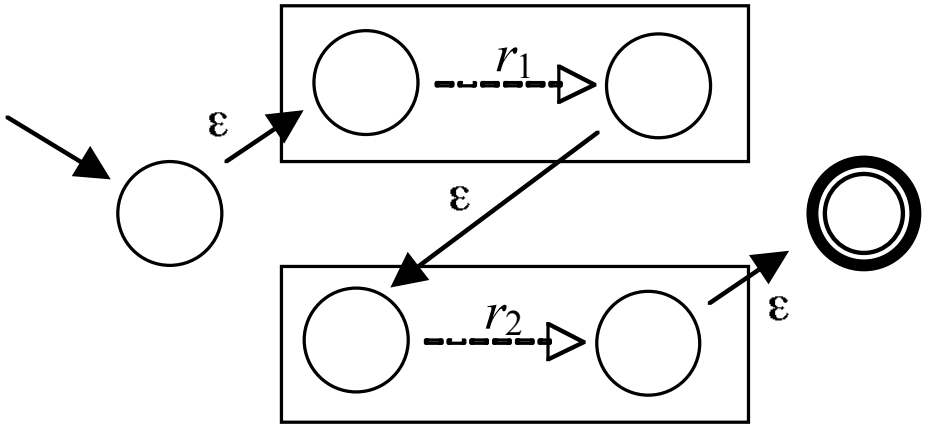
- There are three kinds of *compound* regular expressions:
  - (r_1 + r_2), with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

$(r_1 + r_2)$:



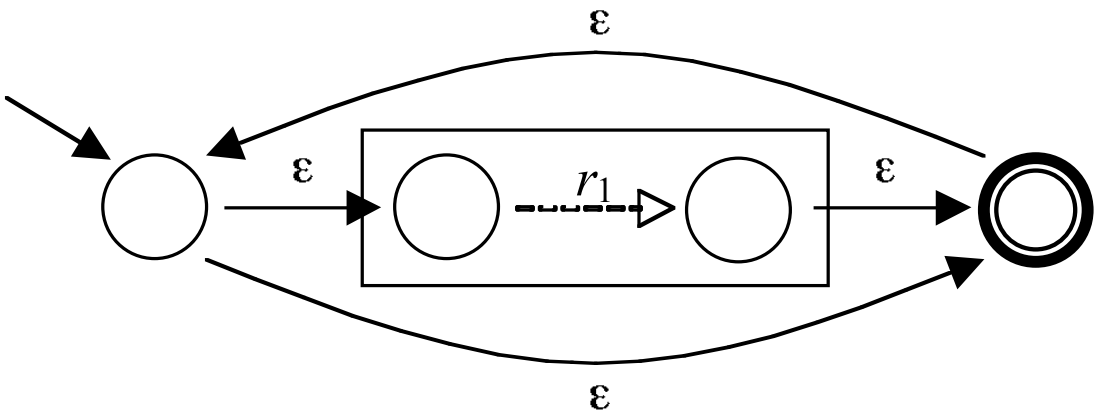
– $(r_1 r_2)$, with $L(r_1 r_2) = L(r_1)\, L(r_2)$

$(r_1 r_2)$:



– $(r_1)^*$, with $L((r_1)^*) = (L(r_1))^*$

$(r_1)^*$:



Outline

- 7.1 Regular Expressions, Formally Defined
- 7.2 Regular Expression Examples
- 7.3 For Every Regular Expression, a Regular Language
- 7.4 Regular Expressions and Structural Induction
- 7.5 For Every Regular Language, a Regular Expression

Sketchy Proof

- That proof left out a number of details
- To make it more rigorous, we would have to
  - Give the 5-tuple form for each NFA
  - Show that it each NFA accepts the right language
- More fundamentally, we would have to organize the proof as an induction: a *structural induction*

Structural Induction

- Induction on a recursively-defined structure
  - Here: the structure of regular expressions
- Base cases: the bases of the recursive definition
  - Here: the atomic regular expressions
- Inductive cases: the recursive cases of the definition
  - Here: the compound regular expressions
- Inductive hypothesis: the assumption that the proof has been done for structurally simpler cases
  - Here: for a compound regular expression *r*, the assumption that the proof has been done for *r*'s subexpressions

Lemma 7.3, Proof Outline

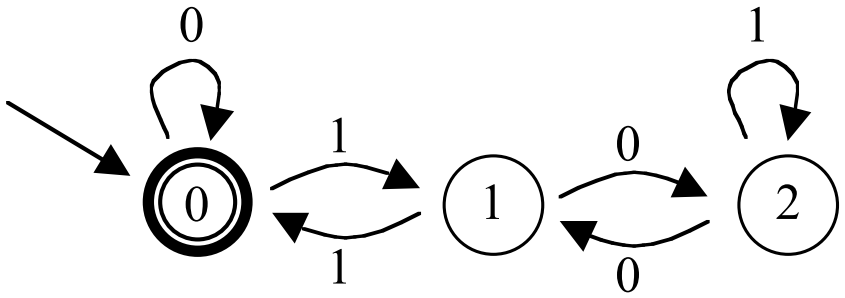
- Proof is by induction on the structure of *r*
- Base cases: when *r* is an atomic expression, it has one of these three forms:
  - *For each, give NFA N and show L(N) correct*
- Recursive cases: when *r* is a compound expression, it has one of these three forms:
  - *For each, give NFA N, using the NFAs for r's subexpressions as guaranteed by the inductive hypothesis, and show L(N) correct*
- QED

Outline

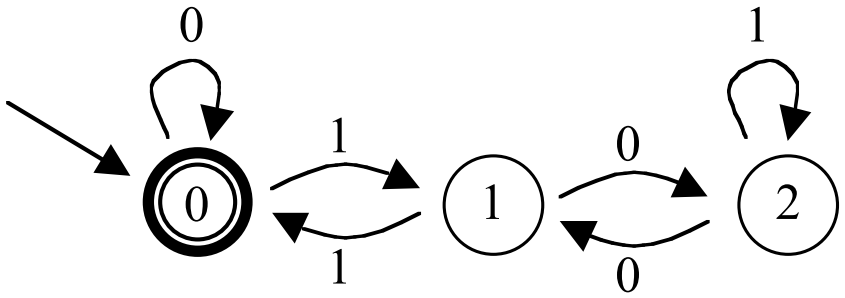
- 7.1 Regular Expressions, Formally Defined
- 7.2 Regular Expression Examples
- 7.3 For Every Regular Expression, a Regular Language
- 7.4 Regular Expressions and Structural Induction
- 7.5 For Every Regular Language, a Regular Expression

NFA to Regular Expression

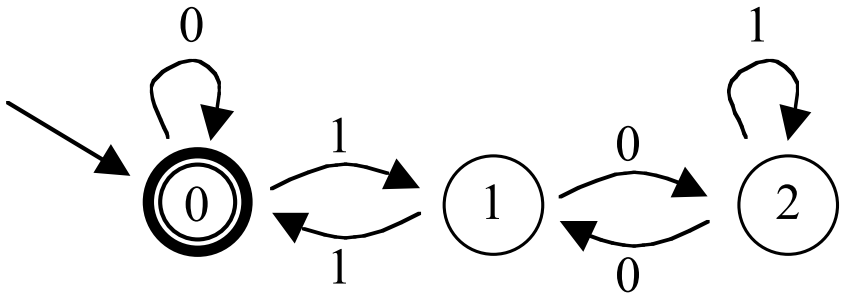
- There is a way to take any NFA and construct a regular expression for the same language
- Lemma 7.5: if N is any NFA, there is some regular expression r with $L(r) = L(N)$
- A tricky construction, covered in Appendix A
- For now, just an example of the construction



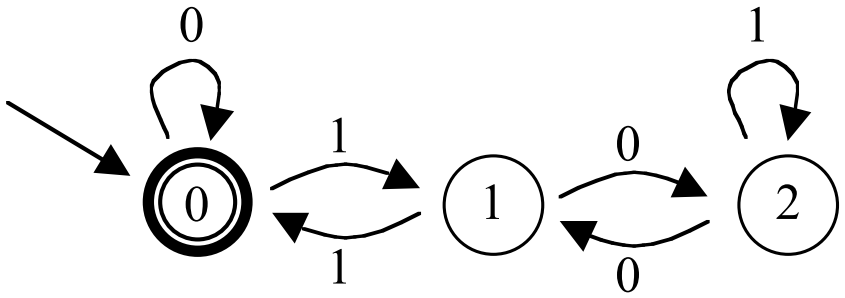
- Recall this NFA (which is also a DFA) from chapter 3
- *L*(*M*) = the set of strings that are binary representation of numbers divisible by 3
- We'll construct an equivalent regular expression
- Not as hard as it looks
- Ultimately, we want the set of strings that take it from 0 to 0, passing through any of the other states
- But we'll start with some easy pieces



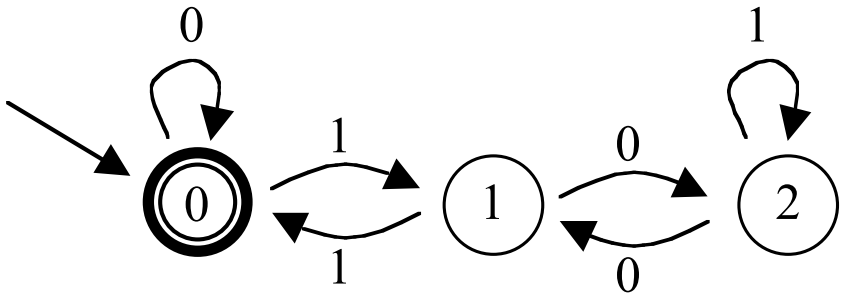
- What is a regular expression for the language of strings that take it from 2 back to 2, any number of times, without passing through 0 or 1?



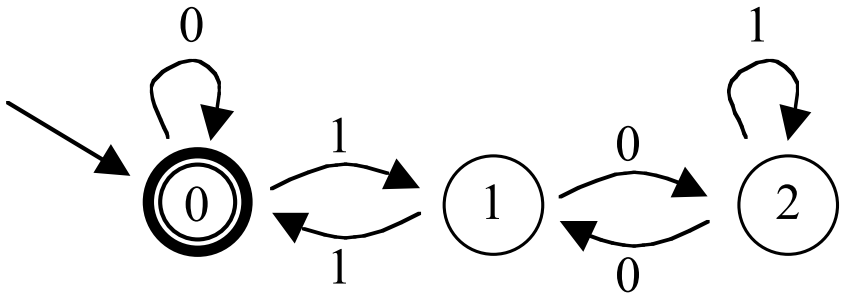
- What is a regular expression for the language of strings that take it from 2 back to 2, any number of times, without passing through 0 or 1?
  - Easy: 1*



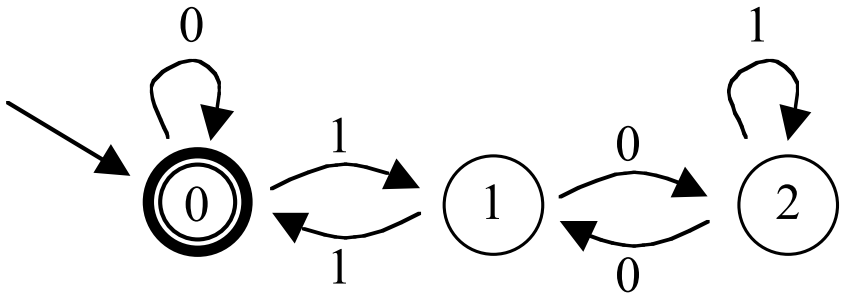
- What is a regular expression for the language of strings that take it from 2 back to 2, any number of times, without passing through 0 or 1?
  - Easy: 1*
- Then what is a regular expression for the language of strings that take it from 1 back to 1, any number of times, without passing through 0?



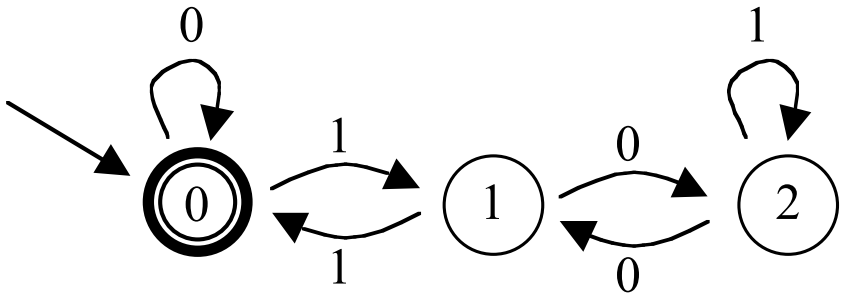
- What is a regular expression for the language of strings that take it from 2 back to 2, any number of times, without passing through 0 or 1?
  - Easy: 1*
- Then what is a regular expression for the language of strings that take it from 1 back to 1, any number of times, without passing through 0?
  - That would be (01*0)*:
    - Go to 2 (the first 0)
    - Go from 2 to 2 any number of times (we already got 1* for that)
    - Go back to 1 (the last 0)
    - Repeat any number of times (the outer (..)*)



- Then what is a regular expression for the language of strings that take it from 1 back to 1, any number of times, without passing through 0?
  - That would be (01*0)*
- Then what is a regular expression for the language of strings that take it from 0 back to 0, any number of times?



- Then what is a regular expression for the language of strings that take it from 1 back to 1, any number of times, without passing through 0?
  - That would be (01*0)*
- Then what is a regular expression for the language of string that take it from 0 back to 0, any number of times?
  - That would be (0 + 1(01*0)*1)*:
    - One way to go from 0 to 0 once is with a 0
    - Another is with a 1, then (01*0)*, then a final 1
    - That makes 0 + 1(01*0)*1
    - Repeat any number of times (the outer (..)*)



- So the regular expression is (0 + 1(01*0)*1)*
- The full construction in Appendix A uses a similar approach, and works on any NFA
- It defines the regular expression in terms of smaller regular expressions that correspond to restricted paths through the NFA
- Putting Lemmas 7.3 and 7.5 together, we have...

Theorem 7.5 (Kleene's Theorem)

A language is regular if and only if it is $L(r)$ for some regular expression r.

- Proof: follows from Lemmas 7.3 and 7.5
- This makes our third way of defining the regular languages:
  - By DFA
  - By NFA
  - By regular expression
- These three have equal power for defining languages